VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Asynchronní události

Asynchronous Events

2014 Bc. Jiří Ševčík

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání diplomové práce

Student: **Bc. Jiří Ševčík**

Studijní program: N2647 Informační a komunikační technologie

Studijní obor: 2612T025 Informatika a výpočetní technika

Téma: Asynchronní události
Asynchronous Events

Zásady pro vypracování:

Cílem práce je nastudovat a porovnat jednotlivé metody asynchronního dotazování a komunikace obecně:
1. Průzkum jednotlivých typů komunikace na různých platformách (obecné: select(), specifické pro OS Linux: poll(), epoll(), specifické pro OS FreeBSD: kqueue, Specifické pro OS Windows: IOCP) a případné další.
2. Jejich podrobný popis včetně porovnání se synchronními metodami komunikace ve vláknech
3. Porovnání jednotlivých metod, výhod/nevýhod, zhodnocení.

Výsledkem práce bude:
1. Podrobný popis jednotlivých metod.
2. Výkonové srovnání jednotlivých metod komunikace alespoň pro OS Linux.
3. Zdůvodněné vysvětlení které z metod jsou pro konkrétné způsoby použití nejvhodnější.

Seznam doporučené odborné literatury:

[1] Lukáš Jelínek: Jádro systému Linux - Kompletní průvodce programátora, EAN: 9788025120842
[2] Michael Kerrisk: The Linux Programming Interface: A Linux and UNIX System Programming Handbook, ISBN-10: 1593272200 | ISBN-13: 978-1593272203
[3] Tim Brecht and et al: accept()able strategies for improving web server performance (2004), IN PROC. 2004 USENIX ANNUAL TECH. CONFERENCE
[4] Louay Gammo and Tim Brecht and Amol Shukla and David Pariag: Comparing and evaluating epoll, select, and poll event mechanisms, In Proceedings of 6th Annual Linux Symposium
[5] Gaurav Banga,Jeffrey C. Mogul, Peter Druschel: A scalable and explicit event delivery mechanism for UNIX, ATEC '99 Proceedings of the annual conference on USENIX Annual Technical Conference

Formální náležitosti a rozsah diplomové práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí diplomové práce: **Ing. Nikola Ciprich**

Datum zadání: 01.09.2013

Datum odevzdání: 07.05.2014

doc. Dr. Ing. Eduard Sojka
vedoucí katedry

prof. RNDr. Václav Snášel, CSc.
děkan fakulty

Souhlasím se zveřejněním této diplomové práce dle požadavků čl. 26, odst. 9 *Studijního a zkušebního řádu pro studium v magisterských programech VŠB-TU Ostrava.*

V Ostravě 16. dubna 2014 .....

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 16. dubna 2014 .....

Rád bych na tomto místě poděkoval vedoucímu mé diplomové práce, panu Ing. Nikolovi Ciprichovi, za odborné rady a věcné připomínky. Dále bych rád poděkoval své přítelkyni Vendule Bezákové za velkou podporu při tvorbě práce a také za pomoc s jazykovou korekturou. Poděkování patří i technickému řediteli firmy Linuxbox.cz Petru Kopeckému za poskytnutí testovacích serverů.

Abstrakt

Tato diplomová práce se zaobírá popisy systémů pro zpracovávání asynchronních událostí. Jednotlivé způsoby jsou zde podrobně popsány a je provedeno porovnání jejich vlastností. V rámci práce byla provedena implementace některých notifikačních způsobů a pomocí generátoru zátěže byly provedeny výkonnostní testy.

Klíčová slova: asynchronní události, vlákna, server, generování zátěže

Abstract

This dissertation deals with describing of systems needed for processing of asynchronous events. The individual methods are thoroughly described and a comparison of their properties is carried out. In the framework of the dissertation an implementation of some notifications modes was conducted and the performance tests were performed using the load generator.

Keywords: asynchronous events, threads, server, load generating

Seznam použitých zkratek a symbolů

API	–	Application Programming Interface
AIO	–	Asynchronous Input Output
AIX	–	Advanced Interactive eXecutive
BSD	–	Berkeley Software Distribution
CPU	–	Central Proccessing Unit
HPUX	–	Hewlett Packard UniX
I/O	–	Input/Output
IOCP	–	I/O Completion Ports
IPv4	–	Internet Protocol version 4
OS	–	Operating system
PID	–	Process identifier
POSIX	–	Portable Operating System Interface
RT	–	Real Time
TCP	–	Transmission Control Protocol
UDP	–	User Datagram Protocol

Obsah

Seznam tabulek

Seznam obrázků

Seznam výpisů zdrojového kódu

1 Úvod

Při vývoji moderních a výkonných aplikací, které jsou určeny pro práci s velkým množstvím různorodých požadavků v reálném čase, je často naráženo na problém týkající se způsobu jak nastalé požadavky zpracovávat. Zda použít tradiční přístup, při kterém je zpracováván aplikací v jedné chvíli pouze jeden konkrétní požadavek, či jich ve stejném čase obsluhovat více zároveň.
Oba tyto způsoby mají své vlastní výhody a nevýhody a softwaroví návrháři jsou tak postaveni před obtížné rozhodnutí, který z těchto způsobů zvolit. Toto rozhodnutí může být pro daný projekt velmi kritické, jelikož špatný počáteční návrh činnosti obslužných metod může způsobit v pozdějších fázích vývoje velké problémy. V horším případě může být díky tomuto špatnému počátečnímu rozhodnutí rapidně ovlivněn výkon požadované aplikace. Proto lze tedy tuto volbu označit za velmi důležitou a měl by na ni být brán velký zřetel.
Druhů aplikací, jenž se zabývají tímto problémem či používají jedno z možných řešení, existuje velké množství. Může se jednat například o grafické rozhraní programu, které čeká na událost, jenž je vytvořena interakcí uživatele či třeba o víceprocesovou komunikaci uvnitř operačního systému. Cílem této práce je popsat možné metody použitelné pro zpracovávání takových požadavků. Popisy jednotlivých návrhů a technik budou do jisté míry obecné, ale největší důraz bude kladen na jejich využití v případě serverových aplikací, jež jsou často uzpůsobeny k obsluze velkého množství souběžně připojených klientů. Díky tomuto popisovanému typu aplikace budou v některých částech popsány i další možné způsoby, jak dosáhnout lepších výkonů nejen díky vhodně zvolenému typu obsluhy, ale i za použití dalších funkcí a voleb.

2 Metody pro obsluhu vícenásobných spojení

Při vytváření návrhu serverové aplikace pro zpracování vícenásobných souběžných spojení lze vycházet ze tří základních architektur – multiprocesové, událostmi řízené a kombinace obou. Jak lze vidět na následujících popisech, první dvě z těchto kategorií poskytují odlišný přístup. Ve třetím, kombinovaném způsobu, jsou spojeny základní principy obou z nich. Pro jednotlivé typy budou uvedeny také jejich různé variace užití a některé servery, jež daný typ používají.
Výběr typu architektury může hodně ovlivnit funkčnost či nefunkčnost dané aplikace. V případě špatně zvoleného návrhu nebo její špatné implementace může často docházet k přetížení serveru, nemožnosti přijímat nová spojení nebo například velmi velké latenci odpovědí. Proto by měl být kladen velký důraz na prvotní analýzu aplikace a na jejím základě by měl být zvolen odpovídající model.

2.1 Multiprocesová architektura

Princip tohoto způsobu vychází z použití jednoho hlavního procesu přijímajícího nová spojení a z procesů podřízených, které tato spojení poté zpracovávají a obsluhují. Díky většímu množství procesů jsou obvykle operace I/O blokující.
Místo procesů můžou být používána i vlákna ve stejném vztahu jako zmíněné procesy, ale z důvodů větší obecnosti zde bude používáno termínů proces a podproces. Vytváření obslužných podprocesů lze rozdělit na dvě kategorie:

- **Proces na požádání - on-demand**: nový podproces je vytvořen pokaždé, když je přijato nové spojení, které je tímto procesem poté obsluhováno. Při velkém počtu souběžných spojení může způsobit problémy popisované v jiných kapitolách.
- **Kolekce vytvořených procesů - pre-forked**: aplikace vytvoří několik procesů (systém je obecně nazýván pooling) a v případě přijetí nového spojení je toto obslouženo některým z vytvořených procesů. Takový systém sice eliminuje nevýhody předchozího způsobu v podobě velkého množství procesů, ale sám obsahuje nedostatky, z nichž lze zde uvést například velké využití paměti v případě malé zátěže. S tímto způsobem se lze setkat třeba u FTPD[1] nebo u modifikace phttpd[2].

2.2 Událostmi řízená architektura

Oproti přechozímu způsobu zde figuruje jen jeden proces, který přijímá a obsluhuje veškerá připojení. I/O operace jsou v této architektuře neblokující. Pro sledování událostí nastalých na přijatých spojeních je použit jeden z několika systémů závislých na použité platformě. Popisem těchto systémů se zabývá několik kapitol této práce.
Tato architektura využívá hlavně tří typů, jejichž podrobnějším vysvětlením se zabývá jedna z následujících kapitol. Jedná se o:

[1] http://www.pureftpd.org/project/pure-ftpd
[2] http://freecode.com/projects/phttpd

- událost na spojení je hlášena pokaždé, když jsou dostupná data – úrovňové hlášení,
- událost na spojení je hlášena jen jednou a poté až po změně stavu – hranové hlášení,
- používání asynchronního I/O.

Využití nachází u thttpd[3], userver[4] nebo ribs2[5].

2.3 Kombinace obou předchozích architektur

Poslední architektura kombinuje prvky obou zmíněných ve více možných variacích. Tento způsob je nejvýhodnější a nejpoužívanější pro servery s velkým počtem spojení. S tímto použitím se lze například setkat v modulech http serverů Apache[34] či Nginx[6].

2.4 Server v jádře systému

Dalším možným typem architektury je ne moc rozšířený způsob implementace serveru přímo v jádru operačního systému. Takový způsob může poskytovat určité výhody, co se týče například rychlosti přenosu dat z disku přímo do sítě či zpracování požadavků od přijatých spojení. Existují zde ale neopomenutelné nevýhody:

- problémy s přenositelností mezi různými typy a verzemi jader,
- v případě problému a pádu takového serveru může být ohrožen celý operační systém,
- bezpečnostní politika – každý proces jádra nemá omezená privilegia[23]

Mezi nejznámější patří pod platformou Linux servery khttpd[7] a TUX[8].

[3] http://www.acme.com/software/thttpd/
[4] http://userver.uwaterloo.ca/
[5] https://github.com/Adaptv/ribs2
[6] http://nginx.org/
[7] http://www.fenrus.demon.nl
[8] http://www.stllinux.org/meeting_notes/2001/0719/tux/index.html1

3 Notifikační systémy

V předchozí kapitole byly zmíněny systémy pro sledování asynchronních událostí. Jelikož jich existuje větší množství, je v této práci pro ty nejdůležitější z nich vlastní kapitola zabývající se principem, použitelností, implementačními detaily a i celkovým zhodnocením a porovnáním s ostatními notifikačními způsoby. Vzhledem k tomu, že větší část z nich je využitelná pod platformou Linux, tak pokud nebude řečeno jinak, popis a použití se bude týkat této platformy. Také použité funkce budou popsány pro jazyk C. V textu je dále zmíněno větší množství systémových volání a funkcí. V některých případech je k dané funkci i podrobnější popis a o ostatních lze získat další informace v příslušné dokumentaci [24] [25].
Výčet všech popsaných systémů jistě nebude zcela kompletní a existují další možnosti, které jsou většinou uzpůsobeny jen pro konkrétní platformu. Na Solaris/HPUX lze třeba použít systém */dev/poll* a pro AIX se nabízí *Event completion* nebo *pollset*.

V jednotlivých kapitolách jsou popsány tyto systémy:

- *select()*
- *poll()*
- *epoll()*
- *kqueue()*
- vlákna
- *fork()*
- RT signály
- *IOCP*

3.1 Frameworky

Aby byl usnadněn vývoj serverových aplikací a předešlo se redundantnímu řešení známých problémů, je možno použít některé z dostupných frameworků. V jejich implementacích lze nalézt již vyřešené a ihned použitelné některé z typů architektur či strategií popsaných v této práci. Jedná se například o:

- **ACE**: obsahuje objektově orientovanou implementaci některých strategií, nabízí velké množství voleb pro neblokující I/O[9];
- **ASIO**: oproti předchozímu obsahuje několik dalších voleb a je součástí knihovny Boost[10];

[9] http://www.cs.wustl.edu/ schmidt/ACE.html
[10] https://www.asio.gov.au/Publications/ASIO-People-Capability-Framework.html

- **libevent**: multiplatformní, podpora všech pěti zde popsaných notifikačních systémů, podpora vláken. Využíván například v aplikacích Chromium, Tor či ntpd[11];
- **libev** – vylepšení libevent[12].

[11] http://libevent.org
[12] http://software.schmorp.de/pkg/libev.html

4 Select

Nejstarším notifikačním systém popisovaným v této práci je mechanismus *select()*, který je jako jediný přenositelný mezi platformami UNIX, Linux a Windows. Ze základního principu funkčnosti systému *select()* byly později odvozeny další notifikační systémy zmiňované v této práci. První implementace se objevila v srpnu 1983 na platformě 4.2BSD Unix.

4.1 Princip

Základním předpokladem pro používání *select()* v uživatelském procesu je mít jeden či více deskriptorů pro sledování. Čísla těchto deskriptorů je poté potřeba pomocí makra *FD_SET* přidat do speciálního vektoru, který je zastupován strukturou *fd_set*. Struktura samotná má podobu bitové masky, kde jsou při přidávání deskriptorů pomocí zmíněného makra nastaveny bity pro jednotlivé deskriptory.
Funkci *select()* můžou být předány celkem tři tyto vektory:

- vektor pro čtení: nastavuje se pro deskriptory, na kterých má být prováděno sledování, zda jsou data dostupná ke čtení;
- vektor pro zápis: provádí se sledování, zda je možno do deskriptorů zapisovat;
- vektor pro výjimky: provádí se sledování, zda na deskriptorech nenastaly speciální výjimky nebo chybové stavy.

Po předání vektorů funkci *select()* je prováděno sledování, zda na některém ze sledovaných deskriptorů nenastala událost. V případě že ano, je volání funkce ukončeno a její návratovou hodnotou je celkový počet nastalých událostí.

Volání funkce může být ukončeno dvěma způsoby:

- ukončení po vyvolání jedné či více sledovaných událostí,
- ukončení po specifikovaném časovém limitu.

Po návratu z funkce uživatelský proces projde všechny sledované deskriptory a pomocí makra *FD_ISSET*, kde parametry jsou deskriptor a příslušný vektor, zjišťuje, zda na tomto deskriptoru nenastala sledovaná událost. Pokud ano, je uživatelský proces zodpovědný za zpracování a obsluhu této události.

```
int select(int n, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, struct timeval *timeout);
```

Výpis 1: Funkce *select()*

Kromě již zmíněných parametrů musí být funkci *select()* předáno ještě nejvyšší číslo sledovaného deskriptoru + 1, jenž značí velikost předávaného bitového vektoru. Časový limit je předávám pomocí struktury *timeval*, která obsahuje dvě hodnoty, jež mají datový typ *long* – první z nich jsou sekundy a druhé mikrosekundy. Díky této struktuře je tedy

možné velmi přesně specifikovat čas, po jaký má funkce *select()* čekat na nastalé události. Pokud je časový limit využit, funkce předanou časovou hodnotu aktualizuje, a tím odečte čas, který uběhl od nastalé události. Lze toho využít například k postupnému čekání, ale tato vlastnost je dostupná jen pro platformu Linux.
Lze se setkat i s takovým způsobem užití časového parametru, kde funkci jsou předány nulové parametry a je nastavena jen časová hodnota pro čekání – jedná se tedy jednoduchý způsob na uspání programu po potřebnou dobu. Toto sice umožňuje standardní funkce *sleep()*, ale v ní není možno použít až tak jemného časového rozlišení jako v případě struktury *timeval*. Lze ale použít funkce *usleep()*, která jako svůj parametr má hodnotu v mikrosekundách – není však dostupná pro všechny platformy[26].

```
struct  fd_set  {
  u_int    fd_count;
  SOCKET fd_array[FD_SETSIZE];
} fd_set;
```

Výpis 2: Struktura *fd_set*

Pro volání funkce *select()* můžou být použity i nulové parametry vektorů v případě, že uživatelský proces potřebuje sledovat jen některé z možných nastalých událostí. Pokud je časový parametr nulový, funkce skončí své volání až po obdržení nějaké události. Lze využít nastavení obou *timeval()* prvků na *0*, což zapříčiní okamžitý návrat z funkce po prvním zjištění událostí.

4.2 Návratové hodnoty *select()*

Návratová hodnota u *select()* tvoří počet nastalých událostí – toto číslo je kladné. Navrácena může být i *0* v případě, že vypršela zvolená doba sledování a nenastala žádná událost. V případě chyby je návratovou hodnotou *-1* a je nastavena příslušná hodnota do *errno*, která odpovídá vyvolané chybě. Může se jednat o:

- *EBADF*: použití neplatného deskriptoru v jednom z vektorů,
- *EINTR*: funkce byla přerušena signálem,
- *ENOMEM*: nelze akolovat paměť pro vnitřní struktury.

4.3 Úrovňové hlášení

Funkce *select()* hlásí nastalé události úrovňově. Tento způsob znamená, že událost bude hlášena do té doby, dokud na ni nebude patřičně reagováno (typicky nebudou přečtena všechna data ze socketu). V jiných notifikačních systémech je možno události hlásit i hranově viz kapitola o systému *epoll()*.

4.4 Přidání a odstranění sledovaných událostí

Vytvoření sledování probíhá následovně:

- je vytvořen jeden či více vektorů pro sledování událostí zastupovaný strukturou *fd_set* (dále *fds*),
- fds je vynulován pomocí makra *FD_ZERO*,
- sledovaný deskriptor (*fd*) je přidán do *fds* pomocí makra *FD_SET*,
- je volána funkce *select()*,
- po ukončení funkce je zkontrolováno, zda na sledovaném deskriptoru byla událost vyvolána pomocí makra *FD_ISSET* a pokud ano, je na ni reagováno,
- pokud je po tomto kroku opět volána funkce *select()*, postup je identický.

Vše lze demonstrovat na jednoduchém příkladu, jenž se nachází v závěrečné příloze této práce. Jak lze z něj vidět vidět, musí být při každém volání *select()* vynulován vektor *fds* a znovu nastaveny potřebné deskriptory pro sledování. Důvodem je zde totiž změna původního vektoru při vyvolané události. Tento způsob tedy zapříčiňuje neustálé kopírování dat mezi uživatelským a jádrovým prostorem, což snižuje efektivitu a výkon aplikace.
Možným způsobem urychlení je mít vytvořený hlavní neměnný vektor sledovaných deskriptorů a při každém dalším volání *select()* předávat pouze jeho kopii. Je tím redukováno neustále volání makra *FD_SET* pro registraci deskriptorů. Pro odstranění deskriptoru z vektoru slouží makro *FD_CLR* a pro kopírování deskriptorů mezi jednotlivými vektory lze využít *FD_COPY*. Toto makro je možno využít jen na platformě BSD.[2]

4.5 Využití signálů

V rámci tohoto rozhraní je poskytována i funkce *pselect()*, která funguje stejně jako *select()*, ale liší se v několika vlastnostech:

- lze díky ní nastavit blokování signálů pro dobu jejího volání,
- používá jako časovou hodnotu strukturu *timespec*,
- neaktualizuje předanou časovou hodnotu.

Funkce *select()* v případě, že je uživatelským procesem zachycen signál, přeruší své vykonávání, její návratovou hodnotou je *1* a *errno* je nastaveno na *EINTR*, která značí přerušené systémové volání. Toto chování není moc zvláštní a je používáno ve více systémových funkcích. V případě aplikace používající odchytávání signálů a zároveň sledování událostí pomocí *select()* může nastat problém souběhu, kterému ale lze předejít pomoci vlastností funkce *pselect()*.
Jako modelovou situaci lze uvést příklad, kdy je používán deskriptor pro komunikaci s podřízeným procesem a je použito jen sledování pro data ke čtení. V případě ukončení podřízené procesu by byl vyvolán signál *SIGCHLD* a proběhlo by jeho odchycení rodičovským procesem pomocí obslužné funkce. Funkce by nastavila proměnnou značící, že byl

- | ? | ? | ? | ? | ? | .. | ? |

  vektor *fd_set fds*

- | 0 | 0 | 0 | 0 | 0 | .. | 0 |

  vynulování *fds* pomocí makra *FD_ZERO(&fds)*

- | 0 | 1 | 0 | 0 | 0 | .. | 0 |

  nastavení sledování deskriptoru 1 pomocí makra *FD_SET(1, &fds)*

- | 0 | 1 | 0 | 1 | 0 | .. | 0 |

  nastavení sledování deskriptoru 3 pomocí *makra FD_SET(3, &fds)*

- | 0 | 0 | 0 | 1 | 0 | .. | 0 |

  stav vektoru při návratu z funkce *select(4, &fds, NULL, NULL, NULL)* při nastalé události na deskriptoru 3

Tabulka 1: Nastavení bitů ve vektoru pro funkci select()[14]

podřízený proces ukončen a má být ukončen i proces rodičovský. Kontrola parametru by se provedla před provedením funkce *select()*. Tato část by se mohlo stát kritickou v případě, že by byl signál doručen v místě, kde byla provedena kontrola parametru, ale ještě nebyla volána funkce *select()* s nenastavenou maximální dobou čekání na události. V tomto případě by tedy funkce čekala nekonečně dlouho, čímž by byl chod programu nenávratně zablokován[27].
Řešením je použití blokování signálů v hlavní smyčce, ve které je čekáno na události a také použití *pselect()* společně s dodatečným parametrem značící masku signálů blokovaných po dobu vykonávání této funkce. Po dokončení funkce *pselect()* přestanou být signály blokovány a dojde k jejich okamžitému obsloužení a může být tedy po tomto návratu hned provedena kontrola na parametr značící ukončení. Blokování signálů může být využito i v jiných případech [13] – toto závisí na konkrétní implementaci uživatelského procesu.

4.6 Limit pro počet deskriptorů

Funkce *select()* je limitována počtem možných sledovaných deskriptorů v rámci struktury *fd_set*. Horní hranicí je v tomto případě hodnota konstanty *FD_SETSIZE*, jenž má hodnotu *1024*. Tudíž nemůže být do této struktury přidán deskriptor s číslem větším, než

[13] Diskuze při zavedení *pselect()* - https://lwn.net/Articles/176911/

je uvedená hodnota. Na různých platformách není po určitém nastavení nijak omezen počet a maximální hodnota používaných deskriptorů, ale užití *select()* je vždy limitováno hodnotou *FD_SETSIZE*.

4.7 Sledované deskriptory

Funkce *select()* byla při své implementaci určena primárně pro sledování deskriptorů socketů nebo rour, ale v průběhu let vznikly nejenom na linuxové platformě další funkce, pomocí kterých lze sledovat i další typy deskriptorů. Těmto funkcím je věnována část v kapitole pojednávající o *epoll()*.

4.8 Implementace

V této části, zabývající se vnitřní implementací *select()*, bude popsán způsob, který je používán na platformě Linux. Pro ostatní platformy se ale tento systém moc neliší a základní myšlenka zůstává stejná.
Každý socket nebo obecně každý file deskriptor, který může být použit pro sledování událostí pomocí *select()*, obsahuje strukturu *wait_queue_head_t* (dále *wqh*), ve které je uložen seznam všech čekatelů na nějakou událost na tomto deskriptoru. V případě vyvolání události deskriptor prochází celý připojený seznam a upozorňuje na událost definovanou funkcí.
Funkce *select()* při svém spuštění nejprve zkopíruje do prostoru jádra všechny vektory deskriptorů předaných jako parametr, ve funkci *max_select_fd* zjistí nejvyšší sledovaný deskriptor a poté pomocí *do_select()* je postupně prochází a pro každý z nich vykonává tyto dvě akce:

- přidá záznam do *wqh* seznamu pro daný deskriptor,
- vrátí seznam událostí, které jsou vyvolány pro tento deskriptor (zjištění událostí je věnován popis níže).

Jestliže byla na jednom či více deskriptorech nalezena událost odpovídající té, které je nastavena v jednom z vektorů pro sledování, je poté v tomto vektoru nastaven odpovídající bit pro deskriptor. Po projetí celého seznamu je v případě nějakých nastalých událostí funkce okamžitě ukončena. Pokud tomu tak není, je zkontrolováno, zda nebylo dosaženo časového limitu specifikovaného parametrem *timeout*. V případě že ano, je funkce ukončena. Jinak je uspán až do dosažení tohoto limitu. V případě, že před vypršením této lhůty nastane nějaká událost na jednom z deskriptorů, je probuzeno uvnitř *select()*, je znovu projet celý seznam sledovaných deskriptorů, nastaveny bity a funkce končí.
Z důvodů korektnosti je v průběhu celého vykonávání udržován seznam všech přidaných záznamů do seznamu *wqh* pro všechny deskriptory, jelikož před svým ukončením musí funkce *select()* všechny tyto záznamy postupně odstranit. Samotné záznamy jsou ukládány do struktury *poll_table*[31].
Jak je vidět, nejvíce času funkci zabere procházení všech deskriptorů. Proto byly provedeny pokusy o redukci a zrychlení procházení jednotlivých deskriptorů[7] [9], které dosáhly lepších výsledků než původní implementace.

4.8.1 Zjištění sledovaných událostí na deskriptorech

V této části je popsán vnitřní způsob sledování událostí, který je společný pro zde zmiňované notifikační systémy. Je zde tedy popsán jen jednou a v následujících kapitolách zabývajících se funkcemi *poll()* a *epoll()* na něj bude v příslušných sekcích odkazováno.
Aby mohly být deskriptory či obecně zařízení sledovány pomocí notifikačních systémů, musí mít naimplementovánu funkci, jejíž prototyp vypadá následovně:

*unsigned int (*poll) (struct file *filp, poll_table *wait)*;

Tato metoda je poté volána pokaždé, kdy uživatelský proces spustí zjišťování událostí pomocí nějakého z nabízených notifikačních systémů.

Metoda se skládá ze dvou částí:

- voláním funkce *poll_wait* je zjišťován aktuální status zařízení,
- je vrácena bitová maska popisující možné operace proveditelné okamžitě na daném zařízení.

Obě dvě tyto operace jsou obvykle velmi jednoduché a vypadají velmi podobně na více zařízeních. Vychází se ale z předpokladu, že pouze dané zařízení může poskytnout informace o svém stavu, a proto musí být provedena speciální implementace této funkce na každém z nich. Bitová maska se může skládat z více typů událostí a jejich kompletní popis bude následovat v kapitole popisující systémy *poll()* a *epoll()*, jelikož ty je oproti *select()* mohou všechny (v případě *poll()* jen některé) zpracovávat.

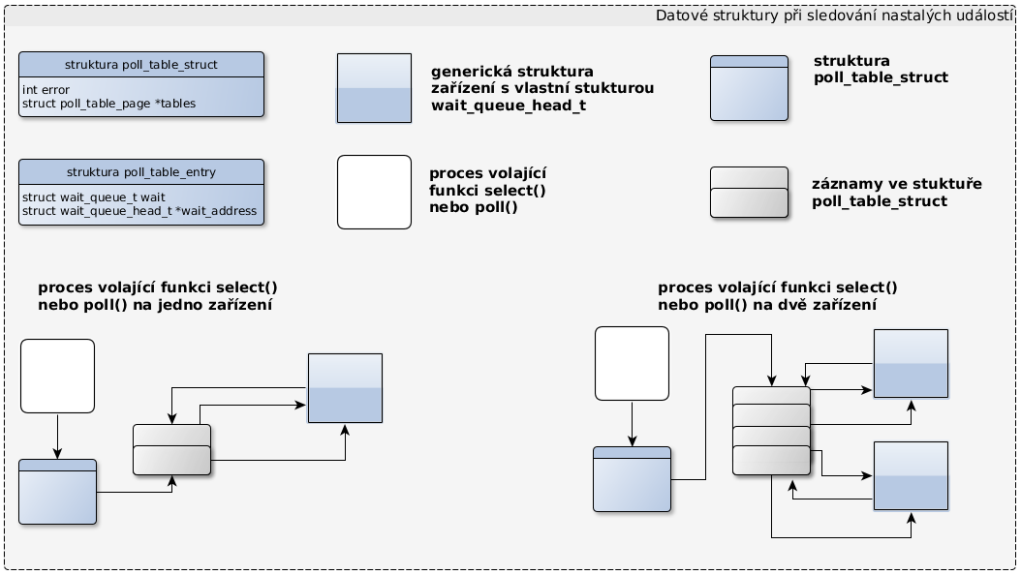
Druhý parametr funkce *poll*, struktura *poll_table*, je používána v rámci jádra pro volání funkcí *select()*, *poll()* a *epoll()* - jsou do ní ukládány záznamy o tom, na jakých zařízení je sledována aktivita. Její deklarace je v hlavičkovém souboru *linux/poll.h* a musí být vložena do zdrojového kódu každého zařízení. Struktura obsahuje seznam záznamů reprezentovaný strukturami *poll_table_entry*, které se navíc skládají ze struktury *file* – označení pro konkrétní sledované zařízení, a z ukazatele na *wait_queue_head_t*. Samotné zařízení s touto strukturou manipuluje jedině pomocí volání funkce

*void poll_wait (struct file *, wait_queue_head_t *, poll_table *)*,

která do *poll_table* přidá záznam *wait_queue_head_t* ze zařízení, které ji volalo. Zajímavou vlastností implementace *poll* je i situace, kdy byly hodnoty pro *timeval()* strukturu nastaveny na *0*. V tomto případě je *poll_table* nastavena na NULL a nejsou vytvářeny jednotlivé záznamy *wait_queue_head_t* u zařízení. A to jednoduše z důvodu, že funkce zjišťuje nastalé události jednorázově a po tomto prvním zjištění bude ukončena a nebude tedy možné čekat na nějakou nastalou událost pro sledované zařízení.[31]. Vazby mezi jednotlivými strukturami jsou lépe viditelné na následujícím obrázku zachycujícím proces s jedním a dvěma sledovanými zařízeními.

4.9 Zhodnocení

Hlavní výhodou této funkce je její přenositelnost mezi více platformami bez použití rozsáhlých úprav, což může být v různých typech aplikací velmi výhodné. Dalším zají-



Obrázek 1: Struktury pro sledování událostí

mavým bodem je jednoduché API. Nevýhod je ale větší počet a většina z nich zde byla zmíněna – kopírování dat mezi uživatelským a kernelovým prostorem a omezení co se týče maximálního počtu deskriptorů. Také je velmi nevýhodným faktem nutnost při každém volání *select()* procházet všechny sledované deskriptory a zjišťovat, zda na některém nenastala událost. Což v případě, že jsou použity všechny tři vektory, může být náročné na výkon – složitost takového postupu je *O(N*3)*. Lze to ale eliminovat pomocí porovnávání návratové hodnoty funkce *select()*, která udává počet nastalých událostí, s počtem již nalezených a obsloužených operací. Vzhledem k tomu, že je zde ale maximální počet deskriptorů striktně limitován, není tedy tato vlastnost tou nejhorší.
Omezující může být i malá škálovatelnost nastalých událostí, jelikož jde využívat jen tří vektorů pro čtení, zápis a výjimky. Poslední nevýhodou vyplývající ze samotné implementace je složitost *O(N)* kdy *N* je počet sledovaných deskriptorů ve vektoru.

5 Poll

Kvůli nedostatkům funkce *select()* byl navržen a naimplementován nový systém *poll()*, který tyto slabé stránky do určité míry eliminuje. Systém byl nasazen poprvé v systému SRV3 Unix vydané v roce 1986 a na platformě Linux se objevil ve verzi 2.1.23 v roce 1997. Oproti funkci *select()* ale není tento systém přenositelný mezi tolika platformami. Následující popis bude opět věnován hlavně použití a implementacím v Linuxu.

5.1 Princip

Základní princip se oproti funkci *select()* v mnohém neliší. Díky eliminaci zmíněných nevýhod jsou změněny struktury pro nastavování sledovaných událostí, a tím je tedy nutno při používání *poll()* vytvářet speciální datové struktury pro každý sledovaný deskriptor a jinak s nimi manipulovat.
Základem je zde struktura *pollfd*, která obsahuje tyto atributy:

- *fd* – file deskriptor sledované události,
- *events* – maska událostí jenž mají být pro daný deskriptor sledovány,
- *erevents* – maska již vyvolaných událostí.

```
struct pollfd {
    int    fd;
    short events;
    short revents;
};
```

Výpis 3: Struktura *pollfd*

Pro každý deskriptor musí být vytvořena tato struktura a nastaveny sledované události. Poté je volána funkce *poll()* (nejedná se ale o funkci se stejným názvem, kterou používá každé zařízení podporující notifikační systémy, viz kapitola o Zjištění sledovaných událostí), jejímž prvním parametrem je celkový počet událostí, ukazatel na pole struktur *pollfd()* a jako třetí je nastavena maximální časová hodnota, po jakou se má provádět sledování. Díky tomu, že je parametrem pole sledovaných událostí a ne velikostně omezený vektor jako u *select()*, není tedy nijak limitován maximální počet sledovatelných aplikací při používání *poll()*[28].

```
int poll(struct pollfd *fds, nfds_t nfds, int timeout);
```

Výpis 4: Funkce *poll()*

Limity můžou být jedině systémového typu jako třeba nastavený maximální počet používaných file deskriptorů v rámci jednoho uživatelského procesu nebo limit celkového možného použití paměti. Jak si lze všimnout, datovým typem parametru po maximální časovou hodnotu čekání na událost zde již není struktura *timeval*, ale *int* nastavující

dobu čekání v milisekundách.
Po svém spuštění funkce kontroluje sledované deskriptory, a jestliže zjistí nastalou událost, v patřičné *pollfd* struktuře nastaví parametr *revents* na bitovou hodnotu této události. V případě, že je na jednom deskriptoru více událostí, jsou jejich hodnoty sloučeny do jedné. Návratovou hodnotou funkce je celkový počet deskriptorů, na kterých nastala jedna či více sledovaných událostí a v případě chyby je nastavena záporná hodnota.
Jakmile je funkce ukončena, uživatelský proces cyklicky projde všechny struktury *pollfd* a pomocí masky nastalých je kontrolováno, o jakou událost se jedná a podle toho je na ni patřičně reagováno.

5.2 Návratové hodnoty *poll()*

Opět jako u *select()* značí kladná návratová hodnota počet deskriptorů s nastalou událostí, *0* žádnou událost a *-1* chybu. Typy *errno* jsou zde:

- *EFAULT*: neplatný parametr fds,
- *EINTR*: vykonávání bylo přerušeno signálem,
- *EINVAL*: parametr nfds překročil limit *RLIMIT_NOFILE* značící maximální možný otevřený počet deskriptorů pro proces,
- *ENOMEN*: nelze alokovat paměť pro vnitřní struktury[28].

5.3 Masky událostí

Nedostatek funkce *select()*, díky kterému bylo třeba při každém dalším volání nutné mazat vektor sledovaných deskriptorů, je zde eliminován právě použitím struktury pollfd, jelikož stačí v dané struktuře masku nastavit jednou a ta poté není při volání *poll()* již nijak modifikována. Modifikace může být ale provedena kdykoliv v uživatelském procesu podle aktuální potřeby pro sledování konkrétních typů událostí. V případě, že některé z těchto událostí nastanou, je jejich bitová reprezentace dostupná v masce *revents*. V případě více nastalých událostí jsou všechny tyto sloučeny do jedné, což pozitivně ovlivňuje výkon.

Jako sledované události můžou být do masky *events* ve struktuře *polld* nastaveny tyto (za názvem události je uvedena její bitová hodnota):

- *POLLIN 0x001* – na deskriptoru jsou dostupná data pro čtení,
- *POLLPRI 0x002* – na deskriptoru jsou out-of-band data [32] ke čtení,
- *POLLOUT 0x004* – do deskriptoru lze zapisovat,
- *POLLRDHUP 0x2000* – socket byl uzavřen nebo bylo uzavřeno spojení před jeho úspěšným dokončením (dostupné od verze jádra 2.6.17).

Dále jsou implicitně pro každou strukturu nastaveny tři další:

- *POLLHUP 0x010* – deskriptor byl uzavřen,
- *POLLERR 0x008* – chyba na deskriptoru,
- *POLLNVAL 0x020* – neplatný deskriptor.

Při kompilaci programu společně s *_XOPEN_SOURCE*[14] může být využito i dalších:

- *POLLRDNORM 0x040*: ekvivalentní s *POLLIN*,
- *POLLRDBAND 0x080*: prioritní data můžou být zapsána (obvykle se na Linuxu nepoužívá),
- *POLLWRNORM 0x100*: ekvivalentní s *POLLOUT*,
- *POLLWRBAND 0x2000*: jsou dostupná prioritní data[28].

Jako další může být na některých platformách použita *POLLMSG* a *POLLREMOVE*.

Lze tedy zpozorovat, že množství sledovatelných událostí je dvojnásobné oproti *select()*, což může být velmi výhodně při potřebě detailnější reakce uživatelského procesu na nastalou událost. V příkladu, jenž je uveden v příloze této práce, je vidět základní použití a nastavení událostí a jejich následné zpracování.

5.4 Hlášení událostí

Stejně jak v případě *select()*, je možno události hlásit pouze úrovňově.

5.5 Využití signálů

Využití blokování signálů popsané v předchozí kapitole lze zde použít úplně jako u použití funkce *ppoll()*. Je zde ale jeden rozdíl v použití této funkce oproti *poll()* - maximální časová hodnota je tu zadávána jako struktura *timespec* (jako v *pselect()*) a ne *int* používaný pro *poll()*.

5.6 Implementace

Celkový princip a způsob implementace *poll()* se moc neodlišuje od způsobu, jakým funguje implementace pro funkci *select()*, a proto zde budou popsány jen některé části, ve kterých se tyto způsoby odlišují:

- do prostoru jádra jsou zkopírovány všechny struktury *pollfd* předané funkci *poll()* jako parametr,
- není zjišťován nejvyšší sledovaný deskriptor,

[14] http://man7.org/linux/man-pages/man7/feature_test_macros.7.html

- procházení jednotlivých struktur je prováděno ve funkci *do_poll()*,
- pro každý sledovaný deskriptor je poté volána funkce *do_pollfd()*, ve které je na začátku inicializována proměnná mask s hodnotou *POLLNVAL* (označení pro neplatný deskriptor). Poté je provedena kontrola, zda je možné na daném deskriptoru provést kontrolu událostí. Pokud ano, je hodnota změněna na *DEFAULT_POLLMASK*. Po provedené kontrole jsou v této proměnné bity pro nastalé události a hodnota proměnné je přiřazena do parametru *revents* ve struktuře *pollfd*,
- při nastalých událostech jsou jim příslušné bity nastaveny v parametru *revents* ve struktuře *pollfd* příslušící ke konkrétnímu deskriptoru s vyvolanou událostí.

Kontrola času pro maximální dobu trvání funkce probíhá stejným způsobem jako u *select()*. Po ukončení funkce jsou nastalé události uloženy ve výše zmíněném parametru. Podobně jako u *select()* byly v minulosti i zde provedeny pokusy o redukci horších vlastností u kopírování sledovaných struktur [9].

5.7 Použití ve Windows

Použití *poll()* na platformě Windows bylo umožněno od verze Vista. Využívá se zde struktury *WSAPollFD* pro sledované deskriptory a jejich masky událostí a funkce *WSAPoll()*[15] pro sledování. Využití sledovatelných událostí je zde podobné jako u *poll()* v Linuxu:

- *POLLPRI*: prioritní data pro čtení (není podporovánu u Microsoft Winsock provider),
- *POLLRDBAND*: out-of-band data pro čtení,
- *POLLRDNORM*: normální data pro čtení,
- *POLLWRNORM*: normální data mohou být zapsána.

Při porovnání s nastavitelnými událostmi u Linuxu lze vidět, že některé z nich chybí, ale dvě z nich lze využít následovně:

- *POLLIN*: kombinace *POLLRDNORM POLLRDBAND*,
- *POLLOUT*: identické jako *POLLWRNORM*.

```
typedef struct pollfd {
  SOCKET fd;
  short events;
  short revents;
} WSAPOLLFD, *PWSAPOLLFD, *LPWSAPOLLFD
```

Výpis 5: Struktura *pollfd* ve Windows

[15] http://msdn.microsoft.com/en-us/library/windows/desktop/ms741669(v=vs.85).aspx

5.8 Zhodnocení

V úvodu bylo řečeno, že *poll()* odstraňuje některé nedostatky popsané u funkce *select()*, což bylo v příslušných částech této kapitoly popsáno a doplněno o další výhody. Je třeba ale zmínit ty nedostatky, které se objevují v obou implementacích, a také ty, jenž jsou jen u *epoll()*.
Vyskytuje se například stejná časová složitost *O(N)* při procházení struktur s jednotlivými sledovanými deskriptory, což může být v kombinaci již dříve popsaného neomezeného použití počtu deskriptorů docela závažným výkonnostním problémem. Dalším problémem je fakt, že je mezi uživatelským a systémový prostorem kopírováno několikanásobně větší množství dat při stejném počtu sledovaných deskriptorů oproti funkci *select()* - v případě *select()* to jsou tři bity pro jeden deskriptor a u *poll()* je to 64 bitů.

6 Epoll

Posledním zmiňovaným notifikačním způsobem využitelným platformou Linux je *epoll()*. Cílem jeho vytvoření byla eliminace všech nedostatků dvou předchozích systémů a také co největší možná škálovatelnost zaručující použití v co nejvíce příkladech.
Tento systém je implementován jen pro platformu Linux a není tedy přenositelný mezi jinými systémy. Jeho zařazení do zdrojových kódů jádra proběhlo v roce 2002.

6.1 Předchozí práce

Hlavní nevýhodou předcházejících funkcí je již několikrát probíraná nutnost při každém volání systémové funkce pro zjištění nastalých událostí kopírovat struktury do paměťového prostoru jádra. O tomto problému se zmiňuje ve své práci Gaurav Banga et al [9]. V rámci tohoto dokumentu byl navržen nový systém založený na dvou hlavních metodách *declare_interest()* a *get_next_event()*. Pomocí první metody by byly do prostoru jádra předávány všechny potřebné struktury s informacemi o sledovaných událostech a to pouze jednou (nebo případně vícekrát při změně například typů sledovaných událostí pro daný deskriptor) a pomocí druhé z nich, *get_next_event()*, by byly vráceny zpět do uživatelského prostoru jen ty struktury, na kterých byla vyvolána událost.
Díky tomuto navrhovanému systému by bylo eliminováno neustálé kopírování dat, jelikož všechny sledované struktury by byly uloženy v prostoru jádra, což by mimo jiné urychlilo jejich zpracování v uživatelském procesu protože by nebylo potřeba procházet úplně všechny sledované struktury, ale jen ty, na kterých byla událost vrácena.

6.2 Princip

Způsob práce s nastalými událostmi je postaven na principu uvedeném v předcházejícím odstavci, který je doplněn o další specifické vlastnosti. Pro použití je třeba na začátku vytvořit speciální strukturu pomocí systémové funkce *epoll_create()*, která vrátí deskriptor pro novou strukturu *epoll*.
Pro jakoukoliv událost, která má být přidána ke sledování, je třeba vytvořit strukturu *epoll_event* a v ní přiřadit do *events* bitovou masku událostí, jenž mají být sledovány. Po tomto přiřazení je nutno strukturu pomocí funkce *epoll_ctl()* přidat ke sledování již vytvořenou strukturou *epoll*, zde zastupovanou deskriptorem. V další částí je již možno čekat na nastalé události pomocí funkce *epoll_wait()* a po jejich vyvolání s nimi pracovat a obsluhovat je.

6.3 Vytvoření deskriptoru pro *epoll()*

Funkce *epoll_create()* po svém korektním vykonání vrátí deskriptor označující nově vytvořenou strukturu *eventpoll* (která bude více popsána v části zabývající se implementací uvnitř jádra) a tento deskriptor je dále používán pro jednotlivé metody. V případě, že již nebude potřeba používat deskriptor, stačí jej uzavřít pomocí funkce *close()*. Toto je provedeno v jádře automaticky v případě, že všechny deskriptory sledované a registrované v

epoll() pro sledování jsou již uzavřeny.
Funkce má jako vstupní parametr hodnotu označující maximální počet deskriptorů, které budou pomocí té instance sledovány – díky tomu si jádro naalokuje potřebný počet používaných vnitřních struktur. Tento způsob byl ale od verze jádra 2.6.8 zrušen a alokace probíhá dynamicky podle průběžného počtu přidávaných událostí. Vstupní parametr ale nadále zůstal (i když není využíván) a jeho hodnota musí být větší než *0*. Funkce lze volat i v její modifikaci *epoll_create1()* s jedním vstupním parametrem. Pokud je tento parametr *0*, funkce se chová stejně jako *epoll_create()*. V jiném případě může být použit parametr *EPOLL_CLOEXEC*, který slouží k nastavení podobného chování jako *O_CLOEXEC* pro funkci *open()*.
Výhodou je, že deskriptor navrácený touto funkcí může být použit ke sledování v jiných notifikačních systémech nebo dokonce i ke generování signálu. Toto použití může být výhodné například v případě, že je potřeba do již existujícího programu, který sám používá některou z předchozích notifikačních funkcí, přidat nový modul využívající právě *epoll()*. V takovém případě je tedy do hlavního procesu předán deskriptor a jakmile je na něm vyvolána událost, je zavolán modul s *epoll()*, který dané události zjistí a případně obslouží nebo předá do procesu hlavního.

6.4 Návratové hodnoty *epoll_create()*

Při úspěšném volání je navráceno kladné číslo pro deskriptor. V případě chyby je vrácena *-1* a hodnota *errno* může být následující:

- *EINVAL*:
  - *epoll_create()*: parametr size není kladné číslo,
  - *epoll_create1()*: neplatný parametr funkce,
- *EMFILE*: překročen maximální limit sledovatelných událostí pomocí *epoll()* , viz Limit,
- *ENFILE*: překročen limit pro počet otevřených deskriptorů,
- *ENOMEN*: problém s pamětí při vytváření struktury *epoll()* v jádře[29].

6.5 Struktura a nastavování sledovaných událostí

Již byla zmíněna struktura *epoll_event*, pomocí které jsou předávány události ke sledování do *epoll()*. Obsah struktury se skládá z parametru *events*, který zastupuje masku událostí pro sledování, a z parametru *data* – jedná se o *union* a lze do něj přiřadit jeden z celkem čtyř různých datových typů. Tento parametr slouží čistě jen pro uživatelská data a jádro s ním nijak nemanipuluje. Nejčastěji je zde ukládán file deskriptor odpovídající události, ale není to samozřejmě podmínkou.

```
typedef union epoll_data
{
  void        *ptr;
  int          fd;
  __uint32_t   u32;
  __uint64_t   u64;
} epoll_data_t;

struct epoll_event
{
  __uint32_t   events;
  epoll_data_t data;
}
```

Výpis 6: Struktura *epoll_event* a *union epoll_data*

6.6 Nastavitelné události

Možné využitelné hodnoty pro events se příliš neliší od těch popsaných u funkce poll(), a proto je zde uveden pouze jejich seznam:

- *EPOLLIN*
- *EPOLLPRI*
- *EPOLLOUT*
- *EPOLLRDHUP*
- *EPOLLHUP*
- *EPOLLERR*

Nevyskytuje se zde událost podobná *POLLNVAL*, která by značila použití neplatného deskriptoru. Toto je v rámci *epoll()* řešeno jiným způsobem, viz popis funkce *epoll_ctl()*.

Oproti poll() je možno nastavit ještě dvě hodnoty, které neznamenají konkrétní událost, ale slouží k upřesnění způsobu, jakým bude hlášení pro daný deskriptor prováděno:

- *EPOLLET*: nastaví hlášení událostí hranově, v další části této kapitoly bude upřesněno
- *EPOLLONESHOT*: při nastalé události bude provedeno jen jedno hlášení a poté bude deskriptor interně deaktivován buď trvale nebo do změny pomocí *epoll_ctl()*.

6.7 Nastavení a modifikace

Po přiřazení hodnot do events je nutné strukturu *epoll_event* zaregistrovat pro sledování, což se provádí pomocí funkce *epoll_ctl()*. Parametry jsou:

- *epfd*: deskriptor pro eventpoll strukturu,
- *op*: typ události, jenž se má provést s parametrem event v rámci epoll systému reprezentovaného parametrem *epfd*:
  - *EPOLL_CTL_ADD*: registrování událostí pro deskriptor,
  - *EPOLL_CTL_MOD*: změna událostí pro deskriptor,
  - *EPOLL_CTL_DEL*: odstranění deskriptoru ze sledování. Odstranění probíhá automaticky v případě, že je deskriptor uzavřen. V případě, že byl deskriptor duplikován (viz níže) a stále v procesu existuje alespoň jeden jeho neuzavřená instance, nejsou automaticky odstraněny všechny s ním spojené struktury a sledování událostí.
- *fd*: deskriptor sledované události,
- *event*: struktura *epoll_event* pro sledovanou událost.

```
int epoll_ctl (int epfd, int op, int fd, struct epoll_event *event);
```

Výpis 7: Funkce *epoll_ctl()*

6.7.1 Návratové hodnoty *epoll_ctl()*

V případě úspěšného nastavení je vrácena *0*, v opačném případě *-1* a je nastavena příslušná *errno*:

- *EBADF*: parametr *epfd* nebo *fd* není validní file deskriptor,
- *EEXIST*: tato chyba nastane v případě, že bylo jako op použito *EPOLL_CTL_ADD* a *fd* je již registrován v rámci této instance. Výjimkou je zde případ, že deskriptor byl duplikován pomocí systémových funkcí *dup()*, *dup1()* nebo *fnctl()*, což muže být výhodné, a to v případě, že pro každý duplikát je nastavena odlišná hodnota *events*.
- *EINVAL*:
  - *epfd* není deskriptor pro epoll instanci
  - *fd* je stejné jako *epfd*
  - zvolený *op* není podporován
- *ENOENT*: jako op bylo použito *EPOLL_CTL_MOD* nebo *EPOLL_CTL_DEL* a zvolený *fd* není registrován v dané *epoll* instanci

- *ENOMEM*: problém s nedostatkem paměti pro provedení operace specifikované v *op*
- *ENOSPC*: dosažen maximální počet sledovaných deskriptorů, viz Limit
- *EPERM*: file deskriptor nepodporuje *epoll()*[29]

6.8 Sledování událostí

Sledování událostí je vyvoláno použitím funkce *epoll_wait()*. V tomto místě je nejzřetelněji vidět rozdílnost *epoll()* oproti *select()* a *poll()*. Jako parametr zde nejsou totiž předávány dříve vytvořené struktury *epoll_events* (které již byly do prostoru jádra zkopírovány jednotlivými voláními *epoll_ctl()*, a díky tomuto kroku byly přidány ke sledování), ale paměťový buffer, který se skládá z jedné či více těchto struktur. Do tohoto bufferu bude *epoll_wait()* ukládat nastalé události. Po návratu z funkce budou v bufferu uloženy jen struktury, na kterých byla vyvolána událost a žádné jiné, což eliminuje zbytečné procházení všech sledovaných deskriptorů známých ze *select()* či *poll()*. Obsloužení jednotlivých událostí již probíhá obdobně v rámci implementace uživatelského procesu. Dalšími parametry *epoll_wait()* je vytvořený deskriptor pro *epoll*, maximální počet událostí vrácených v paměťovém bufferu a také časová hodnota pro maximální dobu čekání (v milisekundách jako u *poll()*).

```
int epoll_wait(int epfd, struct epoll_event *events, int maxevents, int timeout);
```

Výpis 8: Funkce *epoll_wait()*

6.8.1 Návratové hodnoty *epoll_wait()*

Návratové hodnoty jsou identické s již popsanými u *select()* či *poll()*. Rozdílné jsou ale hodnoty uložené v *errno* při nastalé chybě:

- *EBADF*: *epfd* není platný file deskriptor,
- *EFAULT*: oblast paměti určená pro *events* není přístupná pro zápis,
- *EINTR*: volání *epoll_wait()* bylo přerušeno signálem,
- *EINVAL*: *epfd* není platným deskriptorem pro *epoll()* strukturu nebo parametr *maxevents* je menší nebo roven nule[29].

6.9 Využití signálů

I v tomto případě je možné použít modifikovanou funkci pro blokování signálů - *epoll_pwait()*.

6.10 Hranové hlášení událostí

V případě *select()* a *poll()* bylo zmíněno, že hlášení událostí probíhá úrovňově. V případě *epoll()* existuje i další možnost stylu hlášení – hranové. Toto označení, v originále edge-triggered, poprvé použil a specifikoval Jonathon Lemon při popisu notifikačního systému kqueue [15].
Rozdíl v těchto hlášeních tkví v tom, že v případě hranového hlášení je nastalá událost ohlášena při kontrole jen jednou a při dalších kontrolách již ne, a to do té doby, dokud není změněn stav sledované události. Toto lze demonstrovat na jednoduchém příkladu:

1. Server – file deskriptor socketu je zaregistrován do *epoll* pro čtení.
2. Klient – do socketu jsou zapsány 2 kB dat.
3. Server – je volána funkce *epoll_wait()* která vrátí file deskriptor socketu s nastalou událostí pro čtení
4. Server – ze socketu je přečten 1 kB dat.
5. Server – je opět volána funkce *epoll_wait()*.

V tomto místě nastane rozdílné chování při použití hranového hlášení. V případě úrovňového by funkce *epoll_wait()* opět vrátila file deskriptor s událostí pro čtení ale v případě hranového hlášení tomu již tak není. Proces používající hranového hlášení tedy musí přečíst/zapsat všechny data jinak nebude změněna událost na deskriptoru a tedy při další kontrole nebude deskriptor vrácen. Nesprávné použití hranového způsobu hlášení tedy může způsobit nekonečné čekání ve funkci *epoll_wait*. Doporučený způsob užití je tedy při používání neblokujících socketů a při čtení nebo zápisu do té doby, než tyto funkce vrátí konstantu *EAGAIN* značící, že nejsou dostupné již žádné další data. Nastavení *epoll* na hranové hlášení se provádí pomocí funkce *epoll_ctl()* s nastaveným parametrem *EPOLLET*. V základním nastavení *epoll* používá hlášení úrovňové[29].

6.11 Limit

Tento notifikační systém je limitován maximálním počtem zaregistrovaných deskriptorů ve všech vytvořených instancích *epoll()*. Tato limitní hodnota je uložena v */proc/sys/fs/epoll/max_user_watches* a její zavedení se poprvé objevilo ve verzi jádra 2.6.28

6.12 Implementace

Základní princip a vnitřní struktury pro zjišťování událostí přímo na deskriptorech popsaných v kapitole o *select()* je využíván i pro *epoll()*, a proto bude popis jeho implementace hodně vycházet z předchozího textu a nebudou zde již opětovně vysvětlovány principy a významy jednotlivých struktur.
Vzhledem k tomu, že při používání *epoll()* nejsou při každém zjišťování přenášena data do jádra, ale jsou v něm již uložena předtím, jsou vytvořeny pro systém *epoll()* vlastní struktury a funkce pro sledování událostí.

6.12.1 Inicializace

Při vytváření instance *epoll()* je inicializována hlavní struktura *eventpoll*. Mezi její hlavní ukládané parametry patří:

- seznam všech deskriptorů určených pro sledování a jejich odpovídající záznamy. Jednotlivé záznamy jsou reprezentovány strukturami *epitem* a způsob jejich ukládání je založen na principu *red black tree*,
- seznam všech *wait_queue_head_t*, které jsou identické s těmi, které jsou uloženy přímo ve strukturách sledovaných deskriptorů,
- seznam všech deskriptorů, na kterých byla vyvolána událost, která ještě nebyla obsloužena – *rdlist*,
- seznam všech procesů, které v daném okamžiku sledují události nastalé v této struktuře. Tady je využito způsobu, že při vytváření instance *epoll*, což je hlavní struktura *eventpoll*, je navrácen deskriptor. Tento, jak již bylo řečeno, může být použit pro sledování dalšími notifikačními systémy,
- *epitem* struktury obsahující informace o deskriptoru
- informace o uživateli a uživatelském procesu – struktura *user_struct*,
- globální zámky pro práci se strukturou.

6.12.2 Přidání a modifikace

Při volání *epoll_ctl()* pro přidávání či modifikaci událostí pro deskriptor jsou provedeny kontroly na správnost a platnost deskriptoru. Pokud ano, jsou aktivovány zámky, je provedeno hledání, zda již neexistuje v seznamu struktura *epitem* (dále *ep*), jejíž parametry by odpovídaly danému deskriptoru a jeho dalším údajům.
V následujícím kroku jsou podle parametru *op* provedeny přidružené akce odvíjející se od toho, zda byla v předchozí části nalezena odpovídající struktura *epitem* – ano/ne:

- *EPOLL_CTL_ADD*
  - ano: chybová hodnota nastavena na *EEXIST*,
  - ne: do sledovaných událostí jsou přidány implicitně *EPOLLERR* a *EPOLLIN* a *ep* je přidána do seznamu a pro tuto strukturu je nastaven callback pro notifikaci při události.
- *EPOLL_CTL_DEL*
  - ano: *ep* je vyjmuta ze seznamu,
  - ne: chybová hodnota nastavena na *ENOENT*.

- *EPOLL_CTL_MOD*
  - ano: do sledovaných událostí jsou přidány implicitně *EPOLLERR* a *EPOLLIN* a je provedena modifikace sledovaných událostí podle parametrů,
  - ne: chybová hodnota nastavena na *ENOENT*.

V případě nastalého problému a následného nastavení chybové hodnoty jsou zrušeny zámky a funkce je ukončena.

6.12.3 Zjišťování nastalých událostí

Při zjišťování vyvolaných událostí na deskriptorech si lze všimnout více podobných způsobů, jenž jsou později zmiňovány i u popisu *kqueue()*. Vzhledem k rozdílným platformám a jejich notifikačních systémů pro deskriptory bude popis chování proveden jak pro *epoll()*, tak pro *kqueue()* nezávisle na sobě. Je proto možné, že některé části budou podobné.
Pro potřebné zjištění událostí je použita už zmíněná funkce *epoll_wait()*. Po jejím zavolání jsou nejprve provedeny kontroly na validitu předávaných parametrů a poté je předáno řízení funkci *ep_epoll()*.
Zde jsou v první fázi zjišťovány události pro deskriptory. Jestliže jsou nějaké nalezeny, do předaného bufferu jsou nakopírovány jednotlivé struktury a funkce je ukončena. V opačném případě je funkce uspána až do doby, kdy bude vyvolána nějaká událost a funkce bude probuzena, nebo do vypršení časového limitu. V případě událostí je provedeno kopírování jako v předchozím případě.

6.12.4 Notifikace z deskriptoru

Jakmile je na deskriptoru vyvolána událost, je aktivována notifikační funkce pro všechny záznamy ve *wait_queue_head_t*, které jsou přidruženy k tomuto deskriptoru. U *epoll()* se jedná o funkci *ep_poll_callback*, jenž přidá všechny nastalé události do hlavní struktury *eventpoll*. Události jsou zaznamenávány opět jako již vícekrát zmiňované bitové masky. V rámci funkce je i provedeno rušení sledování v případě nastaveného parametru *EPOLLONESHOT*.

6.13 Využitelné deskriptory

Pro dosud zmíněné notifikační systémy lze obecně použít deskriptory, jež mají naimplementovány funkce pro upozornění událostí. Jedná se o sockety, roury či FIFO. Nevýhodou je tedy nemožnost využití sledování například na normálních souborech. Tyto nedostatky byly v rámci implementace dalších funkcí do jisté míry odstraňovány, a tím byla i rozšířena možnost sledování více typů objektů obecně. Dále zmíněné funkce jsou dostupné pro všechny tři notifikační systémy platformy Linux.
Pro všechny z těchto funkcí platí, že po zavolání potřebné funkce je vytvořen deskriptor, jenž je už možno použít pro sledování v některém z notifikačních systému. Jelikož již byly jednotlivé systémy popsány, bude popis přidání a vyvolání událostí jen obecný. Podrobnější popis funkcí je dostupný v dokumentaci[24].

6.13.1 *signalfd()*

Vytváří alternativu pro sledování signálu. Při nadefinování sledovaných signálů je pomocí funkce *signalfd()* vytvořen deskriptor pro sledování. V případě, že je zachycen jeden či více signálů, je na deskriptoru vyvolána událost pro čtení.

6.13.2 *timerfd()*

Vytvoření deskriptoru pro časovač. Může se jednat o časovač jednorázový či opakovaný – specifikuje se podle použité struktury při inicializaci. Po vypršení časového limitu je vyvolána událost pro čtení.

6.13.3 *eventfd()*

Slouží pro vytváření vlastního typu událostí v uživatelském prostoru. Podle vlastní specifikace můžou být poté na deskriptoru vyvolány události pro čtení, zápis nebo chybu.

6.13.4 *inotify*

Tuto funkci lze využít při monitorování událostí ve filesystemu. Podporuje větší počet nastavitelných událostí jako například modifikace souboru, smazání souboru či otevření/zavření souboru. Pro inicializaci je použito několika funkcí, v nichž jsou nastaveny potřebné hodnoty. Jakmile některá z nich nastane, je na deskriptoru vyvolána událost pro čtení.

6.13.5 *aio*

Systém aio umožňuje asynchronní zápis či čtení pro souborové operace. V případě dokončení jedné z možných operací je na deskriptoru opět vyvolána událost ke čtení.

6.14 Zhodnocení

Z popisu možných funkcí a nastavení nad systémem *epoll()* lze jasně vidět, že předčí v těchto ohledech *select()* či *poll()*. Také jsou eliminovány jejich špatné vlastnosti pro neustálé kopírování mezi uživatelským a systémovým prostorem. Je zde sice kopírována větší struktura než v případě vektoru u *select()*, ale toto kopírování je provedeno jen jednorázově. Také zde není tak striktní limit, co se týče počtu sledovatelných deskriptorů. Při využívání i jiných deskriptorů, než jsou například sockety, se tedy jedná o velice komplexní systém, jenž lze uzpůsobit pro velké množství případů při potřebě sledování událostí. Největší nevýhodu lze ale v tomto používání jiných deskriptorů spatřovat ve faktu, že tyto funkce nejsou součástí samotného notifikačního systému, jako je tomu u dále popisované *kqueue()*, je ale třeba používat funkcí jiných, a tím tedy musí být zdrojový kód aplikace o dost komplexnější.

7 Kqueue

Již bylo zmíněno, že implementace kevent byla inspirována u notifikačních implementací pro jinou než linuxovou platformu. Její hlavní inspirací byl systém *kqueue*, jenž je dostupný pro FreeBSD.
Tato platforma může také jako Linux využívat funkcí, jako jsou *select()* a *poll()*, ale z důvodů jejich nevýhod a různých omezení, bylo cílem vytvořit systém, který by:

- nebyl limitován počtem deskriptorů a mohl s nimi ve velkém počtu efektivně pracovat;
- redukce počtu volání systémových funkcí;
- redukce kopírování dat mezi uživatelským prostorem a jádrem;
- nabízel větší flexibilitu co se týče počtu sledovatelných událostí a jejich typů;
- měl jednoduché API a bylo by možno stávající aplikace používající *select()* nebo *poll()* předělat na nový způsob co nejjednodušeji;
- měl být schopen nastalé události hlásit úrovňově i hranově (použití těchto typů hlášení jsou součástí popisu funkce *epoll()*)[15].

Tyto a další vlastnosti a cíle byly základem pro vytvoření notifikačního způsobu *kqueue()*.

7.1 Princip

Při prozkoumání způsobů, jakými kqueue pracuje s událostmi, si nelze nevšimnout faktu, že tyto způsoby jsou hodně podobné principům *epoll()*. Vzhledem k tomu, že implementace *kqueue()* byla provedena o několik let dříve než epoll(), lze usoudit, že *kqueue* pro ni byla v mnohém vzorem.
Kqueue pro svoji činnost musí mít vytvořenou frontu, přes kterou můžou být dále registrovány události ke sledování a přes kterou jsou nastalé události vraceny z prostoru jádra. Pro vytvoření této fronty je použito nového systémového volání *kqueue()*, které v případě úspěšného vytvoření vrátí deskriptor. Ten může být zase použit pro sledování do funkcí *select()* a *poll()* - zde jde opět vidět podobnost s *epoll()* a jeho funkce *epoll_create()*. Zajímavou vlastností je fakt, že pokud proces vytvoří svůj podproces pomocí funkce *fork()*, tak fronta vytvořená voláním *kqueue()* není novým podprocesem zděděna – lze ale využít funkce *rfork()*, která dědičnost v případě nenastavení parametru *RFFDG* umožňuje [16].

V dalším používání se ale kqueue již více liší od jiných způsobů notifikace. Aby mohl být deskriptor či jiný identifikátor zaregistrován ke sledování potřebných událostí, musí být přidán do struktury *kevent*. Přiřazení identifikátoru a ostatních voleb se provádí pomocí makra *EV_SET* – jako i v jiných systémech je možno události a volby kombinovat a nastavovat v jednom volání. Lze tím tedy redukovat počet systémových volání na nejnutnější minimum.

Po vytvoření potřebného počtu těchto struktur jsou předány jako seznam funkci *kevent()*. Zde se vyskytuje jeden podstatný rozdíl oproti ostatním notifikačním způsobům. Funkce totiž jako své parametry obsahuje jednak zmíněný seznam struktur pro sledování, jednak ukazatel na seznam *kevent* struktur, do kterého budou uloženy nastalé události v daném okamžiku volání funkce. Tímto způsobem je také velmi redukován počet systémových volání, jelikož v tomto jednom kroku lze přidávat či modifikovat sledované události a zároveň získat již nastalé události na jiných identifikátorech.
Funkce má dále jako parametry počet přidávaných/modifikovaných událostí, maximální počet návratových struktur s nastalými událostmi a nastavení timeoutu, po jakou dobu má funkce *kevent* čekat na nastalé události. Návratovou hodnotou funkce je počet nastalých a vrácených událostí, který ovšem nemůže být vyšší než definovaný maximální počet. V dalším kroku jsou události obslouženy již způsobem definovaným konkrétním uživatelským procesem jako u jiných notifikačních funkcí.

7.2 Funkce *kevent()*

```
int kevent(kq, changelist, nchanges, eventlist, nevents, &timeout)
```

Výpis 9: Funkce *kevent()*

- *kq* – deskriptor *kqueue*
- *changelist* – události pro sledování (nové či modifikované)
- *nchangelist* – počet událostí v *changelist*
- *eventlist* – návratová struktura pro nastalé události
- *nevents* – maximální počet událostí nastalých událostí
- *timeout* – maximální časová hodnota pro čekání

7.2.1 Struktura kevent

```
struct kevent {
    uintptr_t ident;
    short filter;
    u_short flags;
    u_int fflags;
    intptr_t data;
    void *udata;
};
```

Výpis 10: Struktura *kevent*

7.2.2 Identifikátor - *ident*

Hodnota sloužící pro označení dané události. Interpretace této hodnoty závisí na použitém filtru, ale většinou se jedná o file deskriptor.

7.2.3 Uživatelská data - *udata*

Jedná se o vlastní data, která nemají vliv na žádnou část *kqueue* a jádro do nich také nijak nezasahuje, stejně jako u dat v případě *epoll()*. Může se jednat například o interní identifikátory aplikace pro konkrétní událost nebo o ukazatele na obslužnou rutinu pro danou akci.

7.2.4 Akce - *flags*

Označení operace, která se má provést se strukturou v rámci *kqueue*. Vstupní nastavení:

- *EV_ADD*: přidání události do *kqueue*.
- *EV_ENABLE*: povolení, aby *kevent()* vrátila událost, pokud je tato vyvolána.
- *EV_DISABLE*: zakázání události tak, aby nebyla pomocí *kevent()* vrácena. Filtr samotný zrušen není.
- *EV_DISPATCH*: zakázání události okamžitě po jejím obdržení.
- *EV_DELETE*: odebrání události z *kqueue*. Implicitní chování v případě, že identifikátorem události je file deskriptor, je takové, že je událost smazána automaticky po zavření deskriptoru.
- *EV_RECEIPT*: využitelné při provádění hromadných změn ve struktuře *kqueue*, aniž by byly zrušeny čekající události.
- *EV_CLEAR*: po nastalé události je status resetován, což některé filtry provádějí automaticky. Této volby lze ale využít u těch filtrů, jež reportují přechodné stavy místo stavu stávajícího.
- *EV_ONESHOT*: nastaví, že po první vyvolané události pro zvolený filtr je událost vrácena procesu a poté je automaticky smazána z *kqueue*.

Výstupní nastavení:

- *EV_EOF – End of file*[16] hodnota, jenž je specifická pro použitý filtr.
- *EV_ERROR* – hodnota chyby značící nastalý problém v *kqueue*[16].

[16] http://www.gnu.org/software/libc/manual/html_node/EOF-and-Errors.html

7.2.5 Filtry - *filter*

Nejdůležitější částí, na které je celkový design *kqueue* postaven, jsou filtry. Pomocí filtrů je určováno, jaká událost nastala či nenastala a také lze pomocí jejich nastavení efektivně vracet informace zpět do uživatelského procesu. Způsob používání některých parametrů struktury *kevent* závisí na zvoleném filtru. Potřebné parametry pro tyto filtry jsou předávány pomocí hodnot v parametru *fflags* a data ve struktuře *kevent*.

7.2.5.1 EVFILT_READ Tento filtr vychází z použití čtení z deskriptoru známého z funkcí *select()* a *poll()*. Identifikátorem ve struktuře je zde tedy file deskriptor a událost je vyvolána tehdy, kdy je možno z tohoto deskriptoru číst data. Oproti předcházejícím funkcím ale poskytuje dvě rozšíření:

- v parametru data je při vyvolání událostí vrácena velikost dat, jež jsou připravena ke čtení;
- v případě, že typ deskriptoru má podporu principu *End of file* a tato událost nastane, je nastavena *EV_EOF* hodnota ve filtru a v případě vyskytnuvší se chyby je tato uložena do parametru *fflags*. Je možné, že *EV_EOF* hodnota bude nastavena i v případě, že na deskriptoru jsou data ke čtení – typicky v případě použití socketů.

7.2.5.2 EVFILT_WRITE Jako v předchozím případě je použití tohoto filtru podobné jako v jiných notifikačních systémech – událost je vyvolána v případě, že je možné do daného deskriptoru zapisovat. V parametru data je uložena velikost množství zbývajícího místa v bufferu pro zápis. *EV_EOF* hodnota je nastavena v případě, že se deskriptor odpojil. Tento filtr není podporován pro *vnodes* a *BPF*.
Podporované deskriptory pro filtry *EVFILT_READ* a také pro *EVFILT_WRITE*:

- Sockety – sockety pro síťovou komunikaci. V případě, že se jedná o socket, který byl použit pro *listen()*, je v parametru data vrácena velikost *backlog*.
- *Vnodes*: struktura pro monitorování aktivity nad soubory.
- *Fifo*, *pipes*: kanály pro meziprocesovou komunikaci.
- *BPF*: rozhraní pro plný přístup do linkové vrstvy.

7.2.5.3 EVFILT_AIO Filtr pro používání *aio* – řešení operací souvisejících s asynchronním čtením a zápisem dat. Jako identifikátor je předána struktura *aiocb*.

7.2.5.4 EVFILT_VNODE Jako identifikátor je předán deskriptor souboru nebo složky na disku a sledované události jsou předány pomocí parametru *fflags*. Při vyvolání události je tato vrácena opět pomocí tohoto parametru. Sledovatelné události pro tento filtr jsou (pro větší přehlednost bude v této části při demonstraci možných událostí používán příklad užití na konkrétním souboru, i když se dle předchozího popisu může jednat i o složku):

- *NOTE_DELETE*: bylo použito systémové volání *unlink()* (funkce pro mazání souborů) na soubor.
- *NOTE_WRITE*: do souboru bylo zapisováno.
- *NOTE_EXTEND*: soubor byl rozšířen.
- *NOTE_ATTRIB*: atributy souboru byly změněny.
- *NOTE_LINK*: byl změněn počet odkazů na soubor.
- *NOTE_RENAME*: soubor byl přejmenován.
- *NOTE_REVOKE*: byl zrušen přístup k souboru pomocí funkce *revoke()* nebo byl odpojen celý souborový systém.

7.2.5.5 EVFILT_PROC Využití filtru je pro sledování událostí procesu. Jako identifikátor je do struktury předáno PID procesu. Sledované události jsou opět nastaveny pomocí *fflags* a přes tento parametr jsou v případě vyvolané události i vráceny. Lze sledovat tyto události:

- *NOTE_EXIT*: proces byl ukončen. Návratová hodnota je uložena do parametru data.
- *NOTE_FORK*: byl vytvořen podproces pomocí funkce fork().
- *NOTE_EXECV*: byl spuštěn nový proces pomocí funkce execve() nebo jí podobné.
- *NOTE_TRACK*: v případě, že byl vytvořen podproces, je pro něj vytvořena nová struktura *kevent* s nastavením sledovaných událostí kopírujících rodičovský proces. Podproces poté hlásí událost *NOTE_CHILD* v parametru *fflags* a PID rodičovského procesu je uloženo v parametru *data*. V případě, že se nepodaří vytvořit novou strukturu pro nově spuštěný proces, je vyvolána a do parametru *fflag* uložena událost *NOTE_TRACKKER*.

7.2.5.6 EVFILF_SIGNAL Používá se pro monitorování signálů, které byly poslány do procesu. V případě, že samotný proces již využívá reagování na signály pomocí funkcí *signal()* nebo *sigaction()*, má *kevent* menší prioritu a událost je vyvolána až po předchozích akcích definovaných v těchto funkcích pro práci se signály.
Filtr dále monitoruje všechny pokusy o přístup signálu do procesu, a to i v případě, že signál je procesem samotným blokován (je označen jako *SIG_IGN*). Výjimku zde tvoří *SIGCHLD*, který je vyvolán v případě změny stavu podprocesu[16].
Jakmile je tedy signál zpracován, je vyvolána událost a v parametru data je vrácen počet, kolikrát byl signál vyvolán od posledního volání *kevent()*.

7.2.5.7 EVFILT_TIMER Nastaví libovolný časovač, který je identifikován pomocí parametru *ident*. Časová perioda je specifikována hodnotou v parametru *data* v milisekundách. Událost bude periodicky stále vyvolávána, dokud nebude struktura odstraněna nebo nebude nastaveno *EV_ONESHOT*. Při vyvolání události je v parametru data vrácen počet, kolikrát časovač vypršel od posledního volání funkce *kevent()*[16].

7.2.5.8 EVFILT_USER Vytvoří uživatelsky specifikovanou událost, která není spojena s žádnou funkcí jádra, ale je vyvolána v uživatelském kódu. Parametry jsou zde v bitech nastaveny do *fflags*[16].

7.3 Implementace

7.3.1 Struktury

Základní částí vnitřní implementace *kqueue* je datová struktura *knote*, která koresponduje se strukturou *kevent* jenž je používána v uživatelském procesu. Druhou důležitou strukturou je samotná *kqueue* sloužící ke dvěma účelům:

- zajišťuje seznam obsahující jednotlivé *knotes* s nastalými událostmi, které budou předány do uživatelského procesu;
- sleduje *knotes* odpovídající zaregistrovaným *kevents* pro sledování.

Tyto cíle jsou prováděny pomocí tří dalších podstruktur připojených k hlavní *kqueue*:

- seznam: obsahující *knotes* které byly již dříve označeny jako aktivní (dále jen *kactive*),
- hashovací tabulku sloužící k vyhledání jednotlivých *knotes*, jejichž identifikátor neodpovídá zvolenému deskriptoru,
- lineární pole zřetězených seznamů indexovaných deskriptorů, které jsou přiřazovány stejným způsobem, jako je tomu u procesové open file table[17]. Toto pole je dynamicky rozšiřováno podle momentálně nejvyššího používaného file deskriptoru.

Kqueue musí udržovat seznam všech sledovaných *knotes* z důvodů správného odstranění z paměti v případě, že uživatelský proces uzavře a zruší celou *kqueue*. Dalším důvodem je stav, kdy uživatelský proces uzavře daný file deskriptor a *kqueue* poté odstraní všechny *knotes* přidružené k tomuto deskriptoru[15].

7.3.2 Přidávání sledování událostí

Po zavolání funkce *kqueue()* je v paměti alokována nová struktura kqueue (dále označena jak *kq*), je pro ni vytvořen deskriptor a záznam v open file table. První dvě výše popsané podstruktury nejsou v tomto kroku ještě vytvořeny. Jakmile uživatelský proces zavolá

[17] http://www.cs.kent.edu/ walker/classes/os.f07/lectures/Walker-11.pdf

funkci *kevent()*, seznam změn uložený ve struktuře *changelist*, který je touto funkcí předán jako parametr, je po částech nakopírován do prostoru jádra a následně je každá jednotlivá událost pomocí funkce *kqueue_register()* vložena do *kq*. Tato funkce užívá jako interní identifikátor dvojici parametrů *ident* a *filter* pro prohledání *kq*, zda se již taková událost zde nenachází. Pokud ne (a je-li nastaven parametr *EV_ADD*), je vytvořena nová struktura *knote* (dále jen *kn*). Při této inicializaci je volána rutina pro přiřazení zdroje událostí k této *kq* na základě jejího typu. Dále je provedeno spojení s polem deskriptorů nebo hashovací tabulkou patřící ke *kq*. V případě, že v průběhu nějakého z těchto kroků dojde k chybě, je kód chyby zkopírován do struktury *eventlist*. Teprve jakmile jsou zpracovány všechny záznamy z *changelist()*, je volána funkce *kqueue_scan()* pro vrácení nově nastalých událostí uživatelskému procesu[15].

7.3.3 Rutiny pro filtry

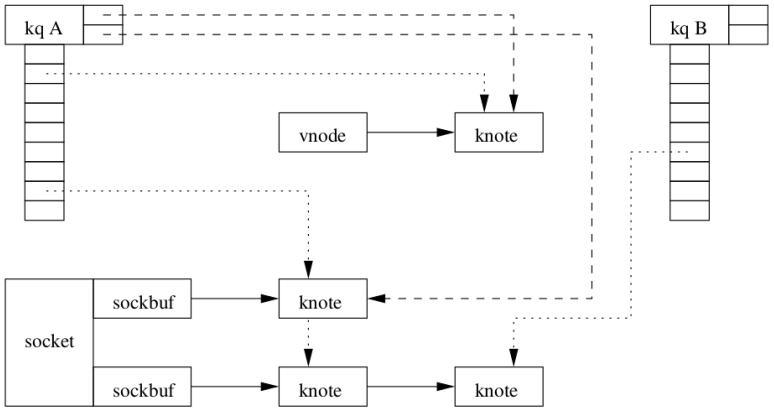
Každý z používaných filtrů nabízí vektor skládající se ze tří obslužných rutin: *attach* (připojení), *detach* (odpojení), *filter*(filtrování).

- rutina pro připojení - *attach*: zajišťuje připojení *knote* struktury k seznamu monitorovaných událostí v nadřazené struktuře *kq*;
- rutina pro odpojení – *detach*: je inverzní k připojovací rutině – v případě zrušení sledování dané události vyjme *knote* ze seznamu sledovaných;
- rutina pro filtrování – *filter*: tato rutina je volána v případě, že byla vyvolána aktivita na přidruženém zdroji událostí. Pomocí rutiny je poté rozhodnuto, zda činnost, která byla vyvolána na zdroji, splňuje podmínku pro hlášení filtrované události, a tudíž vrácení této události do aplikace. Druhy podmínek a jejich vyhodnocování je závislé na použitém filtru pro událost. Samotná funkce musí vrátit booleovskou hodnotu značící, zda sledována událost nastala či nikoliv.

Každá z těchto funkcí kompletně zapouzdřuje informace potřebné k manipulaci se zdrojem událostí, odkud aktivita pochází nebo co reprezentuje. Jedinou informací, kterou nabízí jiným částem implementace *kqueue*, je pouze ta, zda nastala či nenastala sledovaná událost. Díky tomuto jednoduchému zaobalení je umožněno, aby mohly být připojeny další zdroje událostí pouhým přidáním nového filtru.

7.3.4 Aktivita na zdroji událostí

Jakmile je vyvolána událost na sledovaném zdroji, je typicky změněna jeho datová struktura. Ve zdrojových kódech jádra, které tyto možné změny provádí, jsou přidány funkce pro notifikaci existujících *kq*. Pomocí tohoto systému je volána funkce *knot()*. Funkce má jako argument seznam *knotes* pro tento zdroj (ve schématu označený jako *klist*) a jako volitelný parametr jsou zde specifická data pro konkrétní filtr. Dále je funkcí procházen celý seznam a jsou na jeho jednotlivé části volány rutiny o rozhodování, zda nastala sledovaná událost a zda má být provedeno hlášení. Jestliže potřebná událost opravdu nastala, je struktura *knote* přidána na konec seznamu kactive v konkrétní *kq*, ke které je



Obrázek 2: Struktury *kqueue*[15]

tato *knote* přidružena. Přidání je ale provedeno pouze v případě, že se v tomto seznamu ještě nenachází.

7.3.5 Doručení nastalých událostí

Pro vrácení nastalých událostí zpět do uživatelského procesu je použito funkce *kqueue_scan()*. Při jejím volání je na konec seznamu *kactive* přidána speciální *knote* struktura jako označení konce tohoto seznamu pro ohraničení množství struktur, které je třeba zpracovat. Jakmile je značka vyjmuta z *kactive*, je ukončena celá funkce.
Při procházení seznamu je každá knote vyjmuta a proběhne kontrola, zda není nastaven parametr *EV_ONESHOT*. Pokud není, je znovu volána funkce pro filtrování, aby bylo potvrzeno, zda je stav událostí stále validní. Opětovní volání je prováděno pro maximální možné dosažení správnosti a aktuálnosti – může totiž například nastat situace, kdy je již událost v *kactive* listu, ale nebyla prozatím oznámena do uživatelského procesu pomocí *kevent()*. V tomto místě může totiž proces například na socket zavolat funkci *read()*, přečíst všechna data ze socketu, a tím by uložená událost pozbyla platnosti.
Po kontrole a vyjmutí ze seznamu jsou informace z *knote* nakopírovány do *kevent* struktury určené pro návrat do uživatelského procesu. Jestliže je nastaven parametr *EV_ONESHOT*, knote je odstraněna z *kq*. V opačném případě, pokud filtrovací rutina signalizuje, že je událost stále validní, je zařazena na konec seznamu *kactive* – to ovšem za předpokladu, že nemá nastaven parametr *EV_CLEAR*. Tato část probíhá až do nalezení *knote* značky pro konec seznamu, která je pak vyjmuta a do uživatelského procesu je vrácena *kevent* struktura s událostmi[15].

Na schématu je zobrazena *kqueue* A a B společně s jejich seznamy deskriptorů a aktivními knotes. Socket má *klist* pro každý ze svých bufferů a jak je vidět, každý z těchto *klists* může patřit jiné *kqueue*.

7.4 Porovnání s jinými asynchronními způsoby

Z uvedeného popisu možností jasně vyplývá, že v porovnání s ostatními systémy pro zpracování asynchronních událostí je právě *kqueue* nejkomplexnější a možnosti jejího využití jsou daleko širší, než je tomu u ostatních. Tato komplexnost pokrytí velkého počtu typů sledovatelných událostí je sice možná i u jiných systémů používaných v UNIX/Linuxu, ale je k tomu zapotřebí použití dalších dodatečných funkcí. Obrovskou výhodou u *kqueue* je možnost si nadefinovat vlastní událost, což u ostatních implementací není implicitně možné – tato možnost by zapříčinila změnu obrovského množství zdrojového kódu jádra.
Jedinou nevýhodu lze spatřovat u faktu, že *kqueue()* je dostupná pouze pro některé UNIXové platformy. Byl sice proveden prototyp implementace pro Linux[18] či pokus o zavedení nového notifikačního způsobu *kevent()*, ale v současné době nejsou tyto projekty dále rozvíjeny.
Jelikož v rámci této práce byla provedena výkonnostní srovnání jen způsobů použitelných na platformě Linux, pro ilustraci srovnání výkonu byl využit výsledek testů provedený pomocí knihovny *libevent*. Tento test[18] porovnával použití *select()*, *poll()*, *epoll()* a *kqueue* při stu aktivních spojeních. V tomto testu lze jasně vidět, že *select()* a *poll()* dosahoval nesrovnatelně horších výkonů než další dvě metody. Při porovnání výsledků *epoll()* a *kqueue()* jsou již rozdíly daleko menší.

[18] http://alacner.com/book/libevent/libevent-benchmark2.jpg

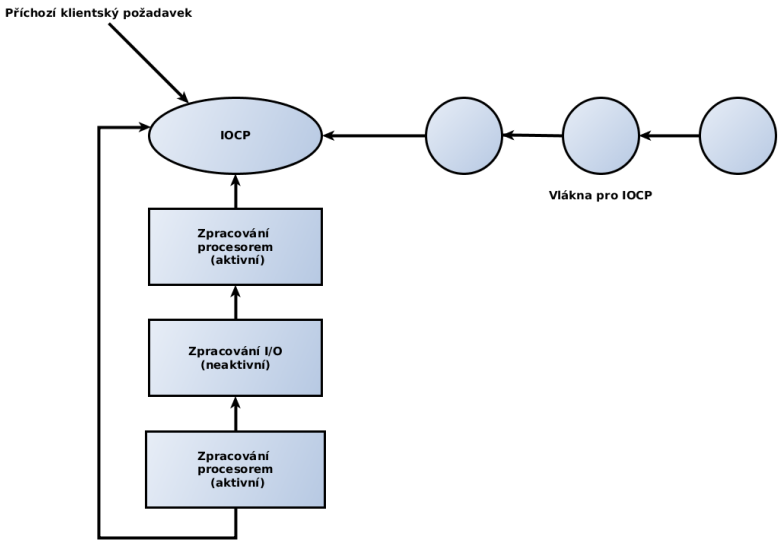
8 I/O Completion Ports

Oproti již zmíněným společným funkcím jako *select()* a *poll()* nabízí platforma společnosti Microsoft Windows speciální rozhraní pro asynchroní operace která se nazývá I/O Completion Ports (dále jen *IOCP*. Toto rozhraní, vyskytující se již od verze Windows NT 3.5, nabízí efektivní vláknový systém pro zpracování četných asynchronních požadavků na multiprocesorových systémech.

8.1 Princip

Aplikace používající *IOCP* vytvoří objekt nazývaný completion port (dále jen *CP*), ke kterému je systémem vytvořena fronta, jejíž účelem je obsluhovat požadavky a vracet informace o dokončeném zpracování. K tomuto vytvořenému *CP* se poté připojí požadované file handlery které využívají asynchroní I/O operace. File handler je spojen s *CP* až do doby, dokud není odebrán.
V případě, že byla nějaká I/O operace dokončena, je zpráva o dokončení (nazývaná též *completion packet*) uložena do fronty principem First In, First Out - FIFO. Operace vyzvedání zpráv o dokončení (v původní implementaci bylo možno vybírat zprávy z fronty pouze po jedné, viz Sekce s popisem API) je prováděna několika vlákny, jejichž maximální počet je specifikován při vytváření *CP*. Vlákno poté vykoná rutinu podle typu zprávy, který značí dokončenou událost. Vlákno tedy přečte/zapíše data, obslouží či přijme nové spojení apod.
Způsob práce vláken je jednou z nejdůležitejších vlastností celého *IOCP*. Aby bylo zamezeno například častému přepínání kontextu (jak je tomu například při použití velkých thread poolů) a jiným náročným činnostem spojeným s vláknovým zpracováváním I/O požadavků, které pro svou činnost vytěžují CPU, má *IOCP* naimplementováno několik vlastností sloužících k co možná nejlepšímu využití systémových prostředků.
První důležitou vlastností je ta, která automaticky po dokončení obslužné rutiny vlákna kontroluje, zda není ve frontě další zpráva pro zpracování. Díky tomuto systému je zamezeno tomu, aby bylo vlákno po dokončení rutiny uspáno, i když jsou další zprávy ve frontě, a pro zpracování požadavku by bylo použito další vlákno, které by muselo být eventuelně probuzeno (jakmile nejsou ve frontě žádné zprávy, tak jsou vlákna uspána). Vlákna jsou tedy poskytována pomocí Last In, First Out - LIFO.

Další a nejdůležitější vlastností je způsob řešení konkurence vláken. Ideálním obecným způsobem při práci s vlákny je mít v daném okamžiku jedno vlákno pro každý procesor v systému. Jak bude dále podrobněji rozebráno, při vytváření *CP* je zvolena hodnota maximálního počtu konkurenčních vláken – tato hodnota tedy omezuje počet zároveň spuštěných vláken spojených s konkrétním *CP*.
V případě, že celkový počet spuštěných vláken pro konkrétní *CP* dosáhne této zvolené hodnoty, systém zablokuje spuštění jakýchkoliv dalších vláken spojených s tímto portem, a to až do té doby, dokud počet vláken neklesne pod zvolenou limitní hodnotu. V rámci *IOCP* je toto řešeno pomocí kontroly hodnoty souběhu – v případě, že hodnota souběhu spojená s konkrétním *IOCP* je rovna počtu aktuálně aktivních vláken, tak další vlákno,



Obrázek 3: I/O completion port operace

které čeká v *CP*, nebude spuštěno.
Výhody tohoto omezení lze nejlépe vidět při nejideálnějším a nejefektivnějším případu při využívání *IOCP*:

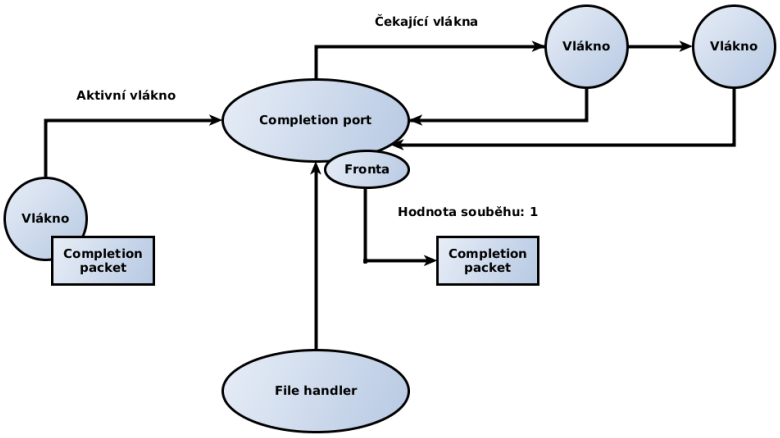
- ve frontě jsou zprávy
- je dosažena maximální hodnota používáných vláken

V tomto případě vlákno, které obslouží jeden požadavek, začne okamžitě obsluhovat další a nedojde tak k žádnému přepínání kontextu, protože běžící vlákno kontinuálně vyzvedává zprávy z fronty a žádná další vlákna nejsou spuštěna – z důvodu, že vlákna jsou distribuována pomocí již zmíněného principu LIFO. Na tomto příkladu je vidět, že další vlákna by byla zbytečná a nikdy by nebyla spuštěna.

8.2 Využitelné file handlery

V rámci *IOCP* lze pro přiřazení k *CP* použít všechny objekty, které využívají I/O metody. Patří sem například soubory na disku, TCP sockety, pojmenované roury nebo třeba i mail sloty. Obecně se jedná o objekty, které mohou využít následující funkce:

- **ConnectNamedPipe** - umožňuje serverové pojmenované rouře čekat na klientský proces připojením nové instance pojmenované roury
- **DeviceIoControl** - poslání kontrolního kódu ovladači zařízení, které poté provede odpovídající operaci přidruženou k danému kódu



Obrázek 4: I/O completion port operace s jedním povoleným vláknem

- **LockFileEx** - vytvoření zámku na daný soubor pro konkrétní proces. Zámek může být i sdílený
- **ReadDirectoryChangesW** - čtení informací o změnách v zadaném adresáři
- **ReadFile** - čtení dat ze specifikovaného souboru a nebo I/O zařízení
- **TransactNamedPipe** - kombinace čtení a zápisu z pojmenované roury do jedné síťové operace
- **WaitCommEvent** - čekání na událost na specifikovaném zařízení
- **WriteFile** - zápis dat do specifikovaného souboru a nebo I/O zařízení
- **WSASendMsg** - zápis dat a volitelných kontrolních informací z připojeného nebo odpojeného socketu
- **WSASendTo** - zápis dat do specifikovaného socketu
- **WSASend** - zápis dat do připojeného socketu
- **WSARecvFrom** - příjem datagramů a uložení zdrojové adresy
- **WSARecvMsg** - příjem dat a volitelných kontrolních informací z připojeného nebo odpojeného socketu
- **WSARecv** - příjem dat z připojeného socketu[1]

8.3 Hlavní funkce

IOCP nabízí větší množství funkcí z nichž nejdůležitější jsou dvě níže uvedené.

8.3.1 CreateIoCompletionPort

Funkce sloužící k vytvoření nového *CP* s možností připojení file handleru a nebo pro připojení file handleru k již existujícímu *CP*.

```
HANDLE WINAPI CreateIoCompletionPort(
  _In_ HANDLE FileHandle,
  _In_opt_ HANDLE ExistingCompletionPort,
  _In_ ULONG_PTR CompletionKey,
  _In_ DWORD NumberOfConcurrentThreads
);
```

Výpis 11: funkce CreateIoCompletionPort

8.3.1.1 Parametry

- **FileHandle**
  File handler nebo lze použít konstantu *INVALID_HANDLE_VALUE*. V případě použití konstanty není k *IOCP* připojen žádný file handler, parametr *ExistingCompletionPort* musí mít hodnotu *NULL* a *CompletionKey* hodnota je ignorována.

- **ExistingCompletionPort**
  Lze použít dvěma způsoby:

  - parametr je dříve vytvořený *CP* – zadaný file handler je spojen s tímto *CP*
  - parametr je *NULL* – je vytvořen nový *CP*.

- **CompletionKey**
  Uživatelsky definovaný klíč pro file handler, který je poté obsažen ve všech *CP packetech*.

- **NumberOfConcurrentThreads**
  Specifikuje počet vláken pro *CP*. Nejlepší maximální hodnotou pro maximální počet vláken přidružených k *CP* je počet procesorů počítače (toto je nastaveno v případě zvolené hodnoty *0*). V případě použití více vláken bude obslouženo více požadavků ve stejné době, ale jejich zpracování potrvá delší dobu, a proto je třeba uvážit případ, kdy je takové použití vhodné.

- **Návratová hodnota**

  - V případě, že *ExistingCompletionPort* parametr byl *NULL* - je vrácen nový ExistingCompletionPort.
  - V případně, že *ExistingCompletionPort* parametr byl validní *CP* – je vrácen tento samý.
  - V případě chyby je vráceno *NULL*.[1]

8.3.2 GetQueuedCompletionStatus

Funkce vyjímá jednotlivé zprávy z *CP* fronty. V případě, že zde není další zpráva, funkce čeká. Pro možnost vyjmout více zpráv najednou lze použít *GetQueuedCompletionStatusEx*.

```
BOOL WINAPI GetQueuedCompletionStatus(
  _In_   HANDLE CompletionPort,
  _Out_  LPDWORD lpNumberOfBytes,
  _Out_  PULONG_PTR lpCompletionKey,
  _Out_  LPOVERLAPPED *lpOverlapped,
  _In_   DWORD dwMilliseconds
);
```

Výpis 12: funkce GetQueuedCompletionStatus

8.3.2.1 Parametry

- **CompletionPort**
  Konkrétní *CP*.
- **lpNumberOfBytes**
  Počet bytů přenesených za dobu vykonávání I/O operace.
- **LpCompletionKey**
  Uživatelsky definovaný klíč.
- **lpOverlapped**
  OVERLLAPED struktura která byla zadáná, když začala I/O operace. Tato struktura obsahuje informace použité v asynchronním nebo překrývaném I/O.
- **dwMilliseconds**
  Maximální doba v milisekundách, po kterou se má čekat na novou zprávu.
- **Návratová hodnota**
  V případě úspěšného vyjmutí zprávy *True*, v případě chyby (např. při vypršení časového limutu zvoleného v parametru *dwMilliseconds*) *False*. [1]

8.4 Porovnání s epoll()

Při porovnání s vlastnostmi funkce *epoll()* je vidět jeden zásadní rozdíl, který se týká způsobu, jakým jsou doručovány notifikace nastalých událostí:

- V případě *epoll()* je událost vyvolána tehdy, když je file handler připraven vykonat požadovanou operaci.
- V případě *IOCP* je notifikace vyvolána tehdy, když je požadovaná operace dokončena. Díky tomuto způsobu je tedy možné *IOCP* emulovat pomocí separovaného thread poolu. Tohoto využívá například aplikace Wine.[19].

[19]Rozhraní umožňující chod aplikací pro Microsoft Windows pod jinými operačními systémy - http://www.winehq.org

8.4.1 Buffery

Další rozdílnou vlastností je to, *IOCP* operace pro čtení a zápis nemusí dokončit své vykonávání hned, a vzhledem k tomu, že tyto operace využívají buffery, tak je nutné, aby až do kompletního vykonání čtení/zápisu zůstaly tyto buffery nezměněny. Z toho vyplývá několik omezení, jako například nemožnost použít lokální (zásobníkové) buffery, jelikož tyto přestanou být validní po ukončení funkce. Také třeba nelze dynamicky realokovat výstupní buffer v případě, že je potřeba odeslat více dat. Řešením je vytvořit nový buffer. Z této vlastnosti vyplývá i fakt, že není možné měnit jakékoliv parametry pro událost. Pokud tedy například je potřeba zvětšit buffer pro čtení, je nutné sledování operace zrušit a vytvořit novou, ale to skýtá další omezení – viz Změna sledovaných operací.

8.4.2 Více typů operací zároveň

Jednotlivé operace vyžadují také *OVERLLAPED* strukturu, která je neměnitelná až do dokončení vykonávání. V případě, že je tedy nutné zároveň číst a zapisovat, je nutné vytvořit pro každou činnost novou strukturu.

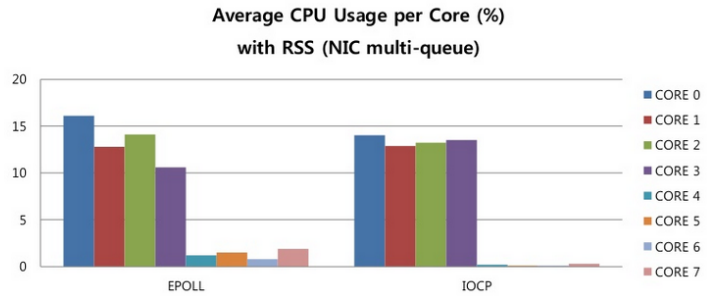
8.4.3 Změna sledovaných operací

U *epoll()* je možné měnit potřebné sledované události jednoduše pomocí funkce *epoll_ctl()*, a to v jakémkoliv vlákně aplikace, aniž by to způsobilo problém pro další vlákna, která čekají na tuto měněnou událost. U *IOCP* je to ale o dost složitější. Na změnu sledovaných událostí lze použít funkci *CancelIo()*, ale může to provést jedině to vlákno, které původně vytvořilo sledování dané události. Status této události je poté neznámý až do doby, dokud není pomocí *GetOverlappedResult()* vrácen status o ukončení.

8.4.4 Větší množství dat

Rozdílné zpracování událostí lze vidět i na situaci, kdy se na socketu vyskytuje větší množství dat a obě funkce vyvolají událost. Jako příklad lze uvést stav, kdy jsou na socketu data ke čtení:

- *epoll()* vyvolá událost *EPOLLIN*, na tu je zareagováno a ve smyčce je možné číst všechna příchozí data až do doby, dokud nějaká na socketu jsou – typicky dokud při čtení není vrácena *EAGAIN*.
- U *IOCP* je ale situace složitější (z již dříve specifikovaného omezení při používání bufferů) a pro příjem dalších dat musí být provedena znovu operace pro čtení, a to do doby, dokud nejsou všechna data přečtena. Což v případě, že pro *IOCP* je používáno více vláken, může vyvolat situaci, kdy více vláken čte zároveň z jednoho socketu a je proto tedy potřeba důkladnější a propracovanější implementace.



Obrázek 5: Porování využití CPU při použití epoll() a IOCP [3]

8.4.5 Neblokující sockety

Při použití neblokujících socketů v *IOCP* je pro připojení socketu nutno použít speciální funkci *ConnectEx()* místo standardní *connect()*, a to z důvodu, že druhá zmíněná nepodporuje *OVERLLAPED* strukturu, a není tedy možné tento socket přidat do *IOCP*. Funkce *accept()* ale již toto omezení nemá.[2]

8.5 Výkonnostní rozdíly

V rámci této práce nebyla výkonnostně testována implementace *IOCP*, ale z již provedených testů výkonu procesoru lze vidět, že při základní implementaci lze dosáhnout relativně stejného výkonu při stejně úspěšném počtu obsloužených spojení. Test byl proveden na serveru s osmi jádry a jako klient byl použit program generující 10000 spojení, z nich každé přijímalo a posílalo na server data po dobu 600 sekund. [3]

8.6 Rozšíření ve Windows

Ve verzi Microsoft Windows 8 and Windows Server 2012 bylo představeno nové rozšíření pro WinSockets – Registered I/O (RIO), které je primárně zaměřeno pro serverové aplikace a používá předvytvořené datové buffery a fronty pro zvýšení výkonu. Podle testů porovnávající implentaci UDP serveru s a bez RIO podpory se jeví využití této nové funkcionality jako více než vhodné. [4]

9 Vlákna

Srovnávání výhod a předností použití čistě jen vláken či událostí je dlouhodobě diskutovaným problémem. Na toto téma bylo v minulosti napsáno velké množství prací přiklánějících se k jednomu či druhému způsobu a vznášející podpůrné argumenty o výhodnosti či Dualita obou systému je zde ještě popsána na grafu popisujícího tok ovládání programu s ohledem na blokování či dosahování kontrolních bodů. V grafu je každý uzel reprezentován jako kontrolní či blokující bod a každá hrana znamená vykonaný kód mezi těmito dvěma body. Při porovnání grafu každého z těchto dvou systémů je řečeno, že jsou identické a tím je také podpořeno tvrzení o dualitě obou systémů.nevýhodnosti obou způsobů. Několik takových tezí bude popsáno i v této kapitole. Také je třeba zmínit, že popisy se týkají způsobů používajících jen vlákna nebo události. Nejsou tedy porovnávány jejich možné kombinace - toto je provedeno v jiné částí práce.

9.1 Dualita

H. C. Lauer a R. M. Needham se ve své práci[35], která vznikla již v roce 1978, zabývali touto problematikou a výsledkem jejich výzkumu je tvrzení, že oba dva zmíněné způsoby nemají vůči sobě výhody či nevýhody a tedy že dosahují stejných výkonů. Tato dualita vycházela z porovnání, jak jednotlivé systémy fungují, což se dá shrnout v následující tabulce. Je třeba zmínit, že v jimi používaných výrazech byl systém založený na událostech nazván jako *message-oriented system* a využívání vláken jako *procedure-oriented system*. V tabulce jsou tato porovnání uzpůsobena na aktuálně používanou terminologii.

Dualita obou systému je zde ještě popsána na grafu popisujícího tok ovládání programu s ohledem na blokování či dosahování kontrolních bodů. V grafu je každý uzel reprezentován jako kontrolní či blokující bod a každá hrana znamená vykonaný kód mezi těmito dvěma body. Při porovnání grafu každého z těchto dvou systémů je řečeno, že jsou identické a tím je také podpořeno tvrzení o dualitě obou systémů.

Z dnešního pohledu je toto porovnání ne úplně přesné, a to hlavně ze dvou důvodů:

- ignorování faktu, že událostmi řízené systémy používají pro svou synchronizaci více pokročilý systém plánování;
- používání globálních datových struktur a sdílené paměti bylo popsáno jako velmi netypické pro událostmi řízený systém.

Celkově lze tedy tvrdit, že události řízené programy nebyly posuzovány v jejich obecnosti a nebyly brány v potaz všechny jejich možnosti a využití.

V závěru jejich práce bylo řečeno, že úspěšné nasazení jednoho z těchto systémů je závislé na tom, nakolik je tento způsob více přirozený pro cílovou aplikaci. Tento obecně přijímaný fakt není vyvrácen ani v jedné z následně popsaných pracích.

Události	Vlákna
Kanály pro zprávy	Názvy procedur
Odeslání zprávy	Volání procedury
Zpožděné odeslání, čekání	Vytváření vláken
Čekání v hlavní smyčce	Čekání na uvolnění zámku
Obslužné funkce pro zprávy	Postup procedury
Čekání na zprávy	Stavové proměnné a signály

Tabulka 2: Události vs. Vlákna podle[35]

9.2 Proč jsou vlákna nebo události špatná idea

Zajímavý pohled na problematiku použití systému událostí nebo vláken přináší dvě práce, z nichž ta pozdější[37] se snaží do jisté míry vyvrátit předpoklady, proč je systém založený na událostech lepší a použitelnější ve většině případech[36]. První práce, příznačně nazvaná „Why thread are bad idea (for most purposes)"[36], popisuje důvody, proč je používání vláken těžší a méně výkonnější a jako alternativu doporučuje používání událostí. Není zde ale zamlčen fakt, že i použití vláken má své výhody ve speciálních případech a také že je jejich použití v těchto případech nezbytné. Jedná se o:

- operační systém - jedno systémové vlákno pro jeden uživatelský proces,
- vědecké aplikace - jedno vlákno pro každé CPU,
- distribuované systémy,
- uživatelské rozhraní,
- databázové systémy.

Nevýhod a špatných vlastností je ale uvedeno daleko větší množství. Lze zmínit například:

- složitý vývoj aplikací používajících vlákna,
- dosažení dobrého výkonu aplikace je obtížné,
- problémy se synchronizací,
- deadlock,
- obtížné debugování,
- porušení abstrakce - moduly programu nelze navrhovat nezávisle na sobě,
- callbackové funkce nemůžou pracovat se zámky vláken,
- malá podpora vláken jak u standardních knihoven (typicky thread-safe problém) tak i u funkcí operačních systému,

- ne vždy je potřeba, aby různé části aplikace běžely souběžně.

Nejsou zde ale zamlčeny obecné problémy při používání událostí, jako nemožnost použití více CPU či problémy při delší době vykonávání obslužných rutin, díky čemuž aplikace při zpracovávání jednoho požadavku nemůže obsluhovat další apod.

V druhé práci, tentokrát nazvané „Why events are a bad idea (for high-concurrency servers)"[37], byly některé z vlastností vláken, které byly první prací označeny za špatné či nedostačující, popsány z jiného úhlu pohledu. Díky tomuto jinému způsobu uvažování nad způsobem používání vláken, a také díky vlastní implementaci podpůrné knihovny, bylo na několika testech ukázáno, že oba způsoby si můžou co do výkonu konkurovat. Zmíněné vlastnosti se dají rozdělit do pěti kategorií:

- výkon,
- řízení běhu programu,
- synchronizace,
- správa stavů,
- plánování.

9.2.1 Výkon

Kritika: *Velká spousta pokusů pro použití vláken pro velkou souběžnost nebyla provedena dobře.*

Tato vlastnost není v druhé práci vyvrácena, ale je zmíněno, že dosavadní špatné výsledky jsou zapříčiněny nedostatečnou implementací samotných vláken a jejich knihoven, a to zejména v použití pro velkou souběžnost. Jako hlavní důvod velké režie a následného slabšího výkonu je uváděn počet operací, které mají složitost *O(N)*, kdy *N* je počet používaných vláken. Druhým důvodem je velké množství přepínání kontextu v porovnání při používání událostí.

9.2.2 Řízení běhu programu

Kritika: *Vlákna mají omezené řízení běhu programu.*

Tento problém podle autorů druhé práce vychází z předpokladu, že programátoři při návrhu aplikací přemýšlí velmi lineárně o řízení běhu programu a snaží se používat více komplexnějších návrhových vzorů.
Podle autorů je ale tento přístup špatný z několika důvodů. Hlavní z nich je tvrzení, že komplikované návrhové vzory se v praxi velmi málo využívají. Sami prozkoumali několik typů serverových aplikací a došli k závěru, že jejich návrhové vzory se dají rozdělit do tří jednoduchých částí: volání/návrat, paralelní volání a používání pipe. Všechny tyto částí můžou ale být pomocí vláken prováděny daleko přirozeněji. Je zde vyslovena i myšlenka, že více komplikované návrhové vzory nejsou používány proto, že je obtížné je používat správně.

9.2.3 Synchronizace

Kritika: *Synchronizace vláken je příliš obtížná a náročná.*

Zmíněna je často uváděná výhoda událostmi řízených systémů - díky kooperativnímu multitaskingu [20] je poskytována synchronizace bez toho, aniž by se runtime systém musel starat o vláknové mutexy či obsluhovat fronty pro čekání atd. Díky další práci [21], na kterou je zde odkazováno, jsou tyto vlastnosti výhodami čistě jen kooperativního multitaskingu a ne událostmi řízených systémů, a proto je možno jejich kladných vlastností využít se stejným výsledkem i při použití vláken.

9.2.4 Správa stavů

Kritika: *Stacky vláken jsou neefektivní způsob jak udržovat aktivní stavy.*

Velké systémy založené na používání vláken často riskují mezi přetečením zásobníku a plýtváním virtuálního adresního prostoru pro velké zásobníky, zatímco událostmi řízené systémy tyto problémy řeší obvykle použitím malého množství vláken a uvolňováním stacku po každém volání obslužné rutiny. Pro řešení tohoto problému u vláken autoři vytvořili vlastní mechanismus, jenž umožňuje dynamické zvětšování stacků.

9.2.5 Plánování

Kritika: *Model virtuálního procesoru zajišťovaný vlákny nutí runtime systém být příliš obecný a neumožňuje manuální plánování.*

Událostmi řízené systémy jsou samy schopny plánování akcí přímo na úrovni aplikace. Díky tomu můžou být v aplikaci prováděny různé optimalizace chování či upřednostňování obsloužení některých událostí před ostatními na základě aktuálního stavu. Zde se autoři odkazují na dříve zmíněnou Lauer-Needham dualitu s tím, že stejný způsob plánování může být díky ní aplikován i u vláken.

9.2.6 Porovnání

Díky těmto argumentům na často zmiňovaná nevýhodná místa při používání vláken bylo dosaženo konstatování, že vláknové aplikace můžou dosahovat minimálně stejného výkonu jako aplikace s událostmi a že tyto nemají žádné velké výhody.

Díky nemožnosti větší škálovatelnosti vláken na uživatelské úrovni jsou ale více používány systémy řízené událostmi. Autoři zmiňují, že tato nevýhoda nevychází z vlastností abstrakce vláken, ale z pozůstatků jejich špatné implementace, což dokládají návrhem a vlastní implementací nového systému.

Na základě tohoto nedostatku byl implementován vlastní systém postavený na knihovně *coro*. Samotný systém zahrnuje dynamické navyšování velikosti stacku či vnitřně překládá blokující I/O požadavky na asynchronní. Tato knihovna byla otestována pro

[20] http://wiki.osdev.org/Multitasking_Systems

[21] A. Adya, J. Howell, M. Theimer, W. J. Bolosky, J. R. Douceur. Cooperative task management without manual stack management. InProceedings of the 2002 Usenix ATC, June 2002.

vlastní server *Knot* a bylo provedeno porovnání s již existujícím serverem *Haboob*, jenž je postaven na událostmi řízeném frameworku *SEDA*. Provedené testy potvrdily tezi, že při lepším používání a úpravě dosavadního systému vláken lze dosáhnout lepších výkonnostních výsledků.

9.2.7 Zhodnocení

Na základě těchto prací lze zjistit důležitý fakt, a to že úspěšný a efektivní server či systém obecně není ani tak závislý na zvolení jen událostí nebo vláken, ale na jejich korektním použití a využití všech jejich výhod pro konkrétní aplikaci. Nicméně, jak lze vidět v současnosti, lepších výkonů a škálovatelnosti lze dosáhnout v případě kombinací vláknového a událostmi řízeného systému dohromady. Jako příklad lze třeba uvést nejrozšířenější open-source HTTP server Apache[34]. Tuto skutečnost lze vypozorovat i ve výsledcích testů popsaných v dalších částech této práce.

9.3 Threadpool

Systémem, jenž byl již několikrát zmíněn, je systém označovaný jako threadpool. Jelikož jeho vlastnosti zde byly již vícekrát popsány, v této části budou zmíněny jen ty, které se jinde nevyskytují. Jeho hlavní nevýhoda nastává při vytvoření několika set souběžných spojeních a korespondujícím počtu vláken. To vychází z faktu, že je každému nově vytvořenému vláknu při jeho spuštění přidělen vlastní stack, který má např. u systému Linux[30] velikost 2 MB, což v případě 500 spojení se rovná 1 GB zabrané v RAM. To sice nemusí být na moderních serverech nijak velké číslo, ale musí být brány v potaz dvě věci:

- na platformě Linux je maximální počet běžících procesů (vlákno je také bráno jako proces se sdíleným adresním prostorem) limitován hodnotou zjistitelnou z */proc/sys/kernel/threads-max*, která je vypočítatelná z vzorce *mempages / (8 * thread stack size / page size)* kde:
  - mempages – celkový počet stránek paměti,
  - thread stack size – velikost stacku vlákna/procesu,
  - page size – velikost jedné stránky paměti.
- Počet používaných vláken pro jeden proces je limitován hodnotou, kterou lze zjistit z rovnice *total virtual memory/ (thread stack size*1024*1024)* kde:
  - total virtual memory – celková virtuální paměť,
  - thread stack size – velikost stacku vlákna[38].

  Tento počet lze ale pro proces navýšit a může nastat situace, kdy serverová aplikace vytvoří obrovské množství vláken a postupně může zabrat veškerou dostupnou paměť'[22].

[22] *Out Of Memory problém*

Zmíněné problémy jsou ještě výraznější v případě, kdy operační systém nepodporuje použití vláken na úrovni kernelu a aplikace tedy musí používat vlákna na úrovni uživatelského procesu.

10 Fork

Další možností pro obsluhu událostí jak u serverové aplikace, tak i u procesu obecně, je možnost využít systémového volání *fork()*. Tímto voláním je vytvořen k hlavnímu procesu (dále rodič) jeho podproces (dále potomek), a tím vzniká hierarchie procesů.
Potomek obdrží své vlastní *PID* a zároveň sdílí paměť naalokovanou rodičovským procesem před voláním *fork()*. Zde je využito principu *copy-on-write* – paměť je sdílená do té doby, dokud jeden z procesů nemodifikuje data. V tom okamžiku je pro proces vytvořena kopie těchto sdílených stránek dat. Díky tomuto systému je značně urychlen proces vytváření nového procesu a zároveň je zabráněno nekonzistenci dat v obou procesech.

10.1 Využití

Díky tomu, že volání a vytváření potomka není v Unixových systémech náročnou operací, lze jej využít ve velké míře, a to i pro serverové aplikace například následujícím způsobem:

- v rodičovském procesu je přijato nové spojení,
- je vytvořen potomek pomocí *fork()*,
- potomek obslouží dané spojení a skončí.

Tento postup může být v mnoha ohledech dostačující a to hlavně pro jednoduché požadavky. Pouze v případě aplikace, přijímající velké množství nových spojení, není jeho použití moc vhodné a to hlavně proto, že při tisíci nových spojení je vytvořeno korespondující množství potomků a je tím velmi zatížen systém. Toto není takovým problémem v případě, že spojení nejsou perzistentní a jejich obsluha je krátká. Ale v typickém stavu serveru, kdy je poměr neaktivních spojení vůči aktivním značně nerovnoměrný, je použití podprocesů neefektivní. Výhodou je ale fakt, že v případě, že jeden z podprocesů selže a ukončí se, není tím ovlivněn proces hlavní.
Použitelnějším a daleko efektivnějším způsobem v případě velkých serverů s mnoha spojeními je model založený na podobné bázi jako dříve popisované systémy – vytvoření určitého množství podprocesů, mezi něž budou příchozí spojení předávána. Na tomto systému je postaven i jeden z modulů webového serveru Apache[34]. Další možností je zapojení událostmi řízených systémů *select()* apod. takovým způsobem, jako tomu bylo např. při jejich využívání s vlákny.

11 RT signály

Další možností pro zpracovávání IO požadavků je použití real time signálů (dále RT signály). Tyto signály jsou podporovány linuxovým jádrem, ale na rozdíl od tradičních používaných signálů překonávají některé jejich omezení.
Jedním z těchto omezení je nemožnost ukládat a registrovat vícenásobné obdržení stejného signálu. Jako příklad lze uvést situaci, kdy proces nastaví blokování určitého signálu a poté mu několikrát stejné či jiné procesy tento stejný signál opět pošlou. Když po nějaké době proces blokování zruší, obdrží tento signál jen jednou, což sice může být v mnoha ohledech užitečné, ale existují i opačné aplikace, kdy je toto omezení limitující a právě tuto vlastnost řeší RT signály. Umožňují zařazovat obdržené signály do fronty a také k nim ukládají informace v podobě struktury siginfo, v níž je uložen kontext, v jakém byl signál obdržen.
Díky této možnosti lze RT signály využít i pro síťovou socketovou aplikaci. K socketům můžou být pomocí několika volání *fnctl()* připojeny jednotlivé signály. Díky tomuto může jádro ukládat do fronty signály, které byly vyvolány událostmi - například že je možno do socketu zapisovat nebo z něj číst či bylo uzavřeno spojení.

V následujícím příkladu je znázorněn způsob, jakým může aplikace s těmito signály pracovat. Aplikace zde blokuje RT signály a pomocí systémového volání *sigwaitinfo()* synchronně vyjímá RT signály z fronty jádra. Použití této funkce odstraňuje potřebu asynchronního zpracování signálu pomocí další přidružení funkce.

```
int sd = accept (...) ;

fcntl (sd, F_SETOWN, getpid());
fcntl (sd, F_SETSIG, SIGRTMIN);

fcntl (sd, F_SETFL, O_NONBLOCK|O_ASYNC);
```

Výpis 13: Připojení signálu

Jakmile je tedy z fronty načten signál a je zjištěno, že se jedná o RT signál, je možno pomocí zmíněné siginfo struktury zjistit deskriptor spojený s tímto signálem. Obsluha události je již pak závislá na uživatelském procesu.

```
sigset_t signals;
siginfo_t siginfo;
int signum, sd;

sigemptyset(&signals);
sigaddset(&signals, SIGRTMIN);
sigprocmask(SIG_BLOCK, &signals, 0);
while (1) {
    Dequeue a signal from the signal queue
    signum = sigwaitinfo(&signals, &siginfo);
    if (signum == SIGRTMIN) {
        sd = siginfo . si_fd ;
```

```
            handle(sd);
        }
```

Výpis 14: Zpracování signálů

11.1 Omezení

Fronta pro obdržené signály je velikostně limitována, a proto v případě, že je již zaplněna, zašle jádro uživatelskému procesu signál *SIGIO* značící přetečení této fronty. Tato situace může nastat hlavně v případě serveru s velkým počtem připojených socketů generujících různé požadavky.
Uživatelský proces by na toto měl reagovat nejlépe přepnutím do některého jiného notifikačního systému, což musí být provedeno co nejlépe, jelikož při chybné implementaci by mohlo dojít k odmítnutí příchozích spojení či ztrátě dat apod. Druhým limitem, který do jisté míry hodně ovlivňuje zaplnění a přetečení fronty, je nemožnost vybírat z fronty najednou více signálů. Toto chování je tedy rozdílné oproti ostatním notifikačním systémům umožňujícím jedním voláním funkce zjistit více událostí na více deskriptorech.

11.2 Možné vylepšení

Popsanými limity se zaobírá práce týkající se škálovatelností linuxových mechanismů [8]. Zmíněn je zde fakt, že způsob zasílání signálu pro každou událost v rámci jednoho deskriptoru může být při velké zátěži značně neefektivní, a proto tedy mimo jiné může docházet k přeplnění fronty pro signály. Jako způsob řešení byl navržen systém nazvaný *signal-per-fd*. Tento systém je postaven na spojení vyvolaných signálů pro jeden deskriptor – na podobném principu funguje například X Windows system používaný pro grafické uživatelské rozhraní.
V případě, že je pro deskriptor vyvolán signál, je zkontrolováno, zda je již pro deskriptor záznam ve frontě. Pokud ne, je zde přidán nový. Pokud ano, je do tohoto záznamu přidána informace o dalším signálu, což značně urychluje další zpracování po vyjmutí signálu uživatelským procesem.

11.3 Výkonnostní testy

V rámci zmíněné práce o vylepšení RT signálů[8] bylo provedeno několik testů porovnávajících výkon RT signálů, jejich upravené verze *signal-per-fd* se systémy *select()* a *poll()*. V konečných výsledcích dosahovaly RT signály lepší latence než tradiční notifikační způsoby. Testy byly provedeny i v práci týkající se implementace phhtpd serveru, který sám o sobě je postaven právě na těchto signálech.

12 Další alternativy pro Linux

12.1 Kevent

Některé z popsaných notifikačních metod pro platformu Linux obsahují pár již zmíněných negativních vlastností týkajících se například omezení v typech sledovaných událostí, nadbytečné volání jádra či menší počet informací o nastalých událostech.
Pro řešení těchto problému byla navržena technologie *kevent*, jejíž princip je inspirován jinými notifikačními způsoby, které ale nejsou dostupné pro platformu Linux. Vývoj *kevent* byl ale zastaven a nebyla ani začleněna do hlavní větve linuxového jádra. Proto zde budou popsány jen její hlavní vlastnosti a základní princip. Jsou dvě možnosti pro získání událostí v uživatelském prostoru. Buď si lze vyžádat konkrétní počet událostí z bufferu anebo do bufferu přistupovat přímo a pak čekat na novou událost. Jádro je v tomto případě občas informováno o zpracování určitého počtu událostí.

12.1.1 Princip

Celé rozhraní je založeno na principu ring bufferu, jenž je alokován v uživatelském prostoru. Do tohoto bufferu jádro vkládá události a samotný uživatelský proces je poté spotřebovává. Má-li buffer dostatečně velkou kapacitu a jsou-li k dispozici nastalé události, tak se lze po delší dobu obejít bez vstupu do prostoru jádra.
Jsou dvě možnosti pro získání událostí v uživatelském prostoru. Buď si lze vyžádat konkrétní počet událostí z bufferu anebo do bufferu přistupovat přímo a pak čekat na novou událost. Jádro je v tomto případě občas informováno o zpracování určitého počtu událostí.
Přidávání, mazání či modifikace událostí vypadá podobně jako v případě *epoll_ctl* – funkce se zde jmenuje *KEVENT_CTL_ADD*, ale rozsah možností a přenášených informací je větší, než jak je tomu v případě *epoll()*. Také je možnost přidávat ke sledování mimo souborových operací například časovače, signály, události pro AIO nebo přímo uživatelské události, jež jsou generovány ve spuštěném procesu.

12.1.2 Zhodnocení

Toto API bylo komunitou dlouze diskutováno[23]. Vzniklo následně několik patchů, které do jádra implementovaly možnosti sledování signálů nebo třeba i timerů pomocí file deskriptorů, a tím tedy rozšířily stávající možnosti sledování událostí v již dostupných systémech. Takže také díky těmto novým funkcím technologie *kevent* tedy nakonec nebyla začleněna do jádra.

[23] Diskuze u http://lwn.net/Articles/233462/ a http://lwn.net/Articles/225714/

12.2 SEND

Další alternativou pro linuxovou platformu je systém *SEND*, který byl implementován Michaelem Ostrowskim v rámci jeho diplomové práce[39] v roce 2000 – tedy ještě před vytvořením systému *epoll()*. Jím implementovaný systém vychází z teze, že událostmi řízené aplikace nejsou skutečně ovládané těmito událostmi a to z důvodu, že operační systém nebo hardware nemá mechanismus pro propagování událostí do aplikace okamžitě při jejím vyvolání.

12.2.1 Princip

SEND systém využívá *POSIX* signálů pro notifikaci o nastalých událostech a také sdílenou paměť mezi jádrem a uživatelským procesem, která je postavena na konceptu kruhového bufferu. Při nastalé události může být uživatelský proces informován o této událost pomocí jediné systémové funkce, jenž je společná pro všechny typy událostí. Proces poté na toto může reagovat a nastalou událost obsloužit nebo čekat na další události a na ty reagovat hromadně.
Události jsou ukládány do zmíněného kruhového bufferu ve struktuře *event*, která vychází z již dříve používané *siginfo*. Díky používání tohoto sdíleného bufferu je eliminováno neustálé kopírování dat mezi uživatelským a jaderným prostorem, jako je tomu například u funkce *select()*.
Také je možné události sdružovat do skupin a pomocí nové funkce *evtctl()* nastavovat, zda má být v případě jejich vyvolání na ně reagováno a pokud ano, tak jakým způsobem. Systém dále nabízí větší množství škálovatelných funkcí pro různé typy událostí.

12.2.2 Zhodnocení

Při testech v rámci porovnání výkonu se systémem *poll()* bylo v některých částech dosaženo lepších výsledků, a to hlavně v oblasti latence a využití CPU. I přes tyto výsledky nebyl *SEND* více rozšířen a nasazen do reálného využití. Rozhodně se ale jedná o zajímavý koncept používání notifikačního způsobu založeného na využívání *POSIX* signálů.

13 Možnosti zvětšení výkonu serverové aplikace

Bylo již řečeno, že základem výkonné aplikace je správný výběr architektury zpracovávající požadavky. Vlastnost je to sice klíčová, ale existuje další množství způsobů a metod, jež můžou rapidně zvýšit výkonnost serveru. Tyto vlastnosti se můžou týkat jak řešení hardwarového, tak softwarového. První způsob začíná od použití výkonnějšího fyzického stroje, až třeba k využití lepších síťových prvků. V následující kapitole bude řeč o možnosti vylepšení softwarového typu. Těchto je sice také velké množství, jako například možné způsoby používání cache, ale většina těchto vlastností neodpovídá povaze a zaměření této práce. Zaobíráno se bude jednou velmi důležitou částí, jenž je podle mnohých tzv. „úzkým hrdlem obsluhy klientských požadavků“. Jedná se o možné způsoby a strategie, jakými lze přijímat nové spojení a také nastavení různých voleb přímo na socketu.

13.1 Princip přijetí spojení

Přijetí nového spojení se obecně skládá z následujících kroků:

- vytvoření serverového socketu a nastavení příslušných parametrů,
- připojení socketu ke konkrétní adrese funkcí *bind()*,
- nastavení socketu na přijímání spojení pomocí *listen()*,
- přijetí nového spojení pomocí *accept()*.

Pro jednotlivé kroky je třeba vybrat a nastavit vícero dostupných parametrů, ale zmíněny zde budou dva pro tuto práci nejzajímavější:

- při nastavování parametrů socketu lze nastavit příznak *SOCK_NONBLOCK*, díky němuž je socket převeden do neblokujícího režimu, což je pro asynchronní komunikaci nezbytným prvkem. Nastavení lze provést i později pomocí *fnctl()*;
- funkce *listen()* má jako svůj parametr *int backlog*, jenž udává maximální velikost fronty čekajících spojení – viz níže.

13.2 Fronta spojení

Aby mohlo být umožněno klientovi komunikovat přímo se serverem, musí být nejprve provedeno navázaní TCP spojení. Tuto činnost zajišťuje mechanismus s názvem TCP three-way handshake. Až po jeho úspěšném provedení jádro přidá socket do fronty spojení čekajících na přijmutí serverem. V případě, že je tato fronta již plná, je spojení zahozeno a komunikace ukončena. Zaplnění může nastat například ze dvou důvodů:

- server není schopen přijímat požadavky tak rychle, jako přicházejí,
- na server je veden útok zvaný *SYN-flood*, pomocí něhož je zaplňována fronta požadavků.

Druhému případu je možno se bránit několika způsoby a jeho větší popis a řešení ochrany je nad rámec této práce, a proto zde bude řešen a popsán problém první. Ten je řešitelný úpravami strategie, jak by měl server spojení přijímat tak, aby výše popsaný problém nenastal. Vhodná strategie je důležitá také z prostého faktu, že jakmile budou příchozí spojení co nejdříve přijata, tak můžou být co nejrychleji obsloužena, a tím bude dosaženo lepší celkové latence serveru.

13.3 Strategie přijímání spojení

V dřívějších implementacích serverových aplikací byl systém přijímání spojení takový, že po vyvolání události server přijal jedno nové spojení, a buď obsloužil události na již dříve připojených socketech, nebo opět volal některou funkci notifikačního systému pro sledování událostí. V případě, že ve frontě čekajících spojení byl více než tento jeden přijatý socket, byla opět vyvolána událost na serverovém socketu a celý cyklus se opakoval, což je velmi nevýhodné mimo jiné z důvodu většího počtu volání notifikačních funkcí.

V práci, která se zabývala škálovatelností serverů[8], byl popsán a naimplementován systém s názvem *multi-accept*, jenž v případě události na serverovém socketu přijímal nová spojení v cyklu do té doby, dokud se zde nějaká vyskytovala (typicky dokud nebylo volání *accept()* ukončeno se zápornou hodnotou a nastavena příslušná *errno*). Díky tomuto způsobu bylo dosaženo v rámci experimentů lepších výsledků než v jiných testovacích případech, ale za cenu většího vytížení CPU. Na základě tohoto nárůstu výkonu byla vznesena myšlenka, že výkonnostní možnosti serveru jde zvýšit právě tímto způsobem. V následující práci[13] zaobírající se tímto problémem bylo zjištěno, že výše popsaný způsob přijímání spojení sice sníží latenci serveru, ale i přesto není tento způsob nejlepší. Po různých experimentech bylo dosaženo poznatku, že daleko lepších výsledků je možno dosáhnout při vyvážení balance mezi časem, v němž server přijímá nová spojení a dobou, kdy obsluhuje již stávající. Testy byly prováděny nastavením maximálního počtu přijatých spojení v jednom cyklu. Výsledky byly porovnány se způsobem přijímání všech spojení a ve většině případů bylo dosaženo při zvoleném limitu lepšího výkonu. Je třeba ale poznamenat, že tento limit je pro každou architekturu (viz popis architektur) serveru jiný a je třeba najít ideální rozložení a maximální hodnoty po každý typ zvlášť.

13.4 Využití SO_REUSEADDR a SO_REUSEPORT

V testovacích případech této práce bylo použito dvou nastavení, která můžou při správném použití ve velké míře ovlivnit možnosti a výkon serveru zpracovávajícího příchozí požadavky. Jedná se o *SO_REUSEADDR* a *SO_REUSEPORT*, které se zde nastavují přímo na socketu serveru.

13.4.1 Sockety

Implementace socketů na jednotlivých operačních systémech vychází z principů implementace určených pro platformu *BSD*. V průběhu vývoje platforem byly rozšířeny o

různé další funkce, ale princip je na všech stejný. Pro porozumění základům vytváření a používání socketů slouží následující úvod.

13.4.2 Princip a základní použití

TCP/UDP spojení je identifikováno podle pětice hodnot - *[protokol, zdrojová adresa, lokální zdrojový port, cílová adresa, cílový port]*. Aby mohl operační systém rozlišit jednotlivé spojení mezi sebou, musí být tato pětice jedinečná pro každé spojení.

Samotné vytvoření socketu a výběr protokolu se provádí voláním funkce *socket()*, výběr zdrojové adresy a portu pomocí funkce *bind()*. Nastavení cílové adresy a portu pomocí *connect()* v případě použití jako klientského socketu a nebo *listen()* pro použití jako serverového socketi. Z principu této práce bude další popis zaměřen na serverový socket.

V případě výběru portu lze použít hodnotu *0*. Výběr portu je pak závislý na systému a jeho používaném rozsahu. Pro výběr zdrojové adresy lze zvolit konstantu *INADDR_ANY*. V tomto případě je socket připojen na všechny adresy na všech lokálních rozhraních po celou dobu existence tohoto socketu, což znamená, že pokud jsou vytvořeny další lokální interface, je socket připojen i na tyto nové. Lze zmínit i to, že u IPv4 má tato konstanta hodnotu *0:0:0:0*, a v případě IPv6 *::*. V případě, že by bylo vytvořeno více socketů se stejnou zdrojovou adresou a portem, čímž je porušena jedinečnost identifikátoru socketu, vytváření skončí s errorovou hodnotou *EADDRINUSE*. Tomuto lze ale předejít pomocí nastavení vlastností socketu, kterými se zabývá další část této kapitoly.

Mimo socketů pro internetovou doménu existují v linuxových systémech i sockety pro lokální doménu, kdy se socket vytváří jako speciální soubor na disku, ale pracuje se s ním jako se socketem. Výhodou je zde mimojiné to, že lze tomuto socketu nastavit přístupová práva jako normálnímu souboru v systému či větší rychlost oproti internetovému socketu[14].

13.4.3 SO_REUSEADDR

Problémy, které mohou nastat při porušení jedinečnosti identifikátoru socketu, lze vyřešit pomocí užití parametru *SO_REUSEADDR*. Stejně jako v případě *INADDR_ANY* se jedná o konstantu. Nastavována je na socket pomocí standardní funkce *setsockopt()*, kterou lze využít pro nastavování jednotlivých vlastností socketu. Pro zjištění nastavených parametrů lze využít její reverzní funkce *getsockopt()*. Jak bylo řečeno, socketová implementace vychází z původního návrhu pro *BSD* a v následující části jsou zmíněny rozdíly *SO_REUSEADDR* mezi jednotlivými platformami.

Jedna z vlastností *SO_REUSEADDR* lze využít při kombinaci s použitím *INADDR_ANY* pro zdrojovou adresu socketu. Při těchto volbách je změněn způsob, jakým jsou vyhodnocovány konflikty při použití stejných adres. Vše lze jednoduše pochopit pomocí následujících příkladů všech možných kombinací:

V prvním sloupci je uvedeno, zda bylo nastaveno *SO_REUSEADDR* či ne. Možnost *ON/OFF* značí, že její využití nemá na výsledek žádný efekt. Nejvýrazněji lze vidět účinek při porovnání čtvrtého a šestého řádku. Bez nastavení *ON/OFF* není *socketB* vytvořen a s

SO_REUSEADDR	socketA	socketB	Výsledek
ON/OFF	192.168.0.1:21	192.168.0.1:21	Error (EADDRINUSE)
ON/OFF	192.168.0.1:21	10.0.0.1:21	OK
ON/OFF	10.0.0.1:21	192.168.0.1:21	OK
OFF	0.0.0.0:21	192.168.1.0:21	Error (EADDRINUSE)
OFF	192.168.1.0:21	0.0.0.0:21	Error (EADDRINUSE)
ON	0.0.0.0:21	192.168.1.0:21	OK
ON	192.168.1.0:21	0.0.0.0:21	OK
ON/OFF	0.0.0.0:21	0.0.0.0:21	Error (EADDRINUSE)

Tabulka 3: Kombinace při použití *SO_REUSEADDR*

nastavením ano. Je to způsobeno tím, že při nastavení je hodnota v *INADDR_ANY*, která je použita pro *socketA*, brána jako wildcard pro všechny lokální adresy, a adresa *socketB* je brána jako konkrétní specifikovaná adresa.

Další vlastností *SO_REUSEADDR* je možnost vytvořit nový socket na adrese a portu, kde se již nachází jiný socket, který je ve stavu *TIME_WAIT*.

13.4.4 SO_REUSEPORT

Rozdíly mezi touto a předchozí volbou nejsou často jasné, a to hlavně díky ne přesně odpovídajícím názvům. Základní myšlenka pro využití *SO_REUSEPORT* spočívá v tom, že lze připojit více serverových socketů na stejnou adresu a port. Podmínkou, která zabraňuje, aby jiná aplikace nemohla eventuelně zachycovat (omylem či cíleně) některé z příchozích spojení pro jinou aplikaci, je to, že musí být opět použita funkce *setsockopt()* a jako parametr je jí předána konstanta *SO_REUSEPORT*. Další serverové sockety, které chtějí používat stejnou adresu a port, musí poté tento parametr také použít.

Použití této volby ale neznamená, že budou používány i vlastnosti *SO_REUSEADDR* – typickým případem je použití *SO_REUSEADDR* na první socket, který je v *TIME_WAIT* stavu a na druhý socket je použit *SO_REUSEPORT* (adresa i port stejná u obou socketů). Připojení druhého socketu selže, což lze vyřešit dvěma způsoby:

- na prvním socketu nastavit *SO_REUSEPORT*
- využít možnost kombinace obou voleb a na druhém socketu použít obě

Jako další zabezpečení pro zachycení jinou aplikací musí mít všechny další servery využívající tuto adresu a port stejné *UID* jako první server.

Tato volba byla přidána do *BSD* implementace buď později než *SO_REUSEADDR*, a proto není dosud používána v ostatních implementacích, a nebo byla přidána až současné době, viz poslední část této kapitoly.

13.4.5 Rozdíly mezi platformami

- **Linux**

Podpora *SO_REUSEPORT* se v Linuxu objevila až ve verzi 3.9 (datum vydání: 28. 4. 2013). Samotná implementace byla provedena pomocí série patchů od Toma Herberta, který v prvních debatách zmínil, že pracuje s aplikacemi, které přijímají 40000 spojení za sekundu a není tedy překvapením, že autor pracuje u firmy Google.[21] Jako dva hlavní stávající problémy uvedl:

  - v případě použití jediného vlákna se socketem přijímající všechna příchozí spojení, se tato část může stát v extrémních případech kritickým místem aplikace, kdy rychlost vyřízení požadavků je limitována tím, že server není schopen přijímat nová spojení rychleji.
  - druhým problémem byl fakt, že v případě použití více vláken přijímajících požadavky o spojení nejsou při velké zátěži tyto požadavky rovnoměrně rozdělovány mezi jednotlivá vlákna. Ve společnosti Google při testování došli k výsledku, že rozdíly mezi nejméně a nejvíce vytíženým přijímacím vláknem je až trojnásobný, což způsobuje nedostatečné vytížení jader procesoru.

  Samotná implementace dosud obsahuje jeden nedostatek – v případě změny počtu naslouchacích socketů může nastat situace, že příchozí spojení se během průběhu 3-way handshake přeruší, a to z toho důvodu, že při přijetí prvního packetu jsou požadavky automaticky spojeny s přijímacím socketem. Na řešení tohoto problému se ale pracuje.[22]

- **FreeBSD/OpenBSD/NetBSD**
  Tyto platformy jsou forky původního *BSD*, a proto všechny používají stejný způsob jako BSD.

- **MacOS X**
  Založen na forku pozdějšího *BSD*, a proto také využívají stejný způsob.

- **iOS**
  Jedná se o modifikované jádro *MacOS X* a použití je tedy stejné jako v předchozím případě.

- **Android**
  I když jádro tohoto systému je odlišné od většiny Linuxových distribucí, platí pro něj to samé, co ve vztahu *iOs – MacOS*.

- **Windows**
  Používá pouze volbu *SO_REUSEADDR*, která ale používá nastavení stejné jako při použití *SO_REUSEADDR* a *SO_REUSEPORT*, ovšem s tím rozdílem, že může být vytvořen socket na stejnou adresu a port i v případě, že tento nemá nastavenu hodnotu *SO_REUSEADDR*. Tato možnost se jeví jako velice nežádoucí například z důvodu, že jedna aplikace může používat port aplikace jiné, a také se vlastně jedná o bezpečnostní díru. Proto byla na této platformě přidána možnost zaručující výhradní vlastnictví pro socket na konkrétní adrese a portu *SO_EXCLUSIVEADDRUSE*.

- **Solaris**
  Umožňuje použít pouze volbu *SO_REUSEADDR*.

14 Testy

14.1 Testovací server

Aby bylo možné otestovat některé strategie a možnosti pro zpracovávání událostí, byl v rámci této práce naimplementován server *asynCServer*. Server poskytuje několik volitelných parametrů, pomocí kterých je nastavena potřebná metoda pro obsluhu spojení.

14.1.1 Volitelné parametry

- *h*: nastaví příslušnou metodu obsluhy dělící se do 4 kategorií:
  - *s*: *select()*,
  - *p*: *poll()*,
  - *e*: *epoll()*,
  - *t*: *thread()*.
- *t*: výběr konkrétního typu pro zvolenou metodu viz Testovací metody
- *f*: výběr testovacího souboru:
  - *1*: 1B soubor,
  - *10*: 10kB soubor.

Další nastavení jsou uložena v souboru *config*, který je přiložen ke zdrojovým souborům. Jedná se o:

- *server_address*: adresa serveru,
- *server_port*: port serveru,
- *backlog*: velikost fronty spojení,
- *sz_size*: velikost hlavičky,
- *read_buff*: velikost bufferu pro čtení dat,
- *max_clients*: maximální počet přijatých spojení,
- *accept_limit*: maximální počet přijatých spojení v jednom cyklu, viz Strategie přijímání spojení,
- *logpath*: cesta k logovacímu souboru,
- *filePath*: cesta k testovacím souborům,
- *thread_count*: počet použitých vláken.

14.1.2 Vlastnosti

- lze zvolit logování do konzole nebo logování do souboru,
- http odpověď, která je posílána jednotlivým spojením, je vytvořena a namapována do paměti ihned po spuštění serveru a je po celou dobu činnosti neměnná a statická,
- server po obdržení signálu *SIGUSR1* zaloguje info o ukončování a je poté ukončen,
- v případě nastalých chyb je server automaticky ukončen – například nemožnost vytvořit a nabindovat serverový socket, nemožnost namapovat odpověď pro připojení či načtení nevalidních hodnot z *config* souboru,
- je přiložen *Makefile* pro zkompilování. Z důvodů použití *SO_REUSEPORT* lze zkompilovat jen od verze linuxového jádra 3.9 a výš.

14.2 Testovací skripty

Pro porovnání jednotlivých notifikačních možností a jejich kombinací bylo provedeno celkem 1364 jednotlivých testovacích případů, z nichž byl každý celkem třikrát opakován. Díky tomuto velkému počtu by bylo silně nepraktické spouštět klientskou a serverovou stranu samostatně a po každém spuštění ukládat výsledky, a proto byly pro spolehlivou automatizaci a uložení výsledků použity skripty napsané v programovacím jazyce Python. Díky těmto skriptům bylo tedy možno jednotlivé testovací případy spustit postupně za sebou pomocí napsané konfigurace a po dokončení testů bylo možno dosažené a zaznamenané výsledky jednoduše zpracovat do porovnávacích grafů a tabulek. Skripty byly rozděleny na dvě části – klientskou část generující zátěž a serverovou část spouštějící server a zaznamenávající zátěž.

14.2.1 Klientská část

14.2.1.1 Konfigurace Vzhledem k odlišným parametrům jednak pro server a jednak pro generátory zátěže u jednotlivých testovacích případů popsaných v předchozí části byl pro každý test vytvořen konfigurační soubor obsahující potřebné nastavení. Konfigurační soubor obsahuje i možnost nastavit parametry pro testovací nástroj *httperf*, ale jak již bylo zmíněno, nebylo ho pro tuto práci využito. Pro jednoduché použití byla konfigurace uložena jako formát *JSON* a má níže popsanou strukturu. Veškeré konfigurační soubory jsou také obsaženy v datové příloze práce.

```
{  "TestSet": "S1",
   "tests":
   [
      {
         "name" : "wn",
         "server" : "-hs -t1" ,
         "params" :
         [
```

```
            { "name": "01", "idle" : null, "wrk" : "-c 100 -t 2 -d 20s --timeout 10s", "
                httperf" : null },
            { "name": "02", "idle" : null, "wrk" : "-c 200 -t 2 -d 20s --timeout 10s", "
                httperf" : null },
        ]
    }
  ]
}
```

Výpis 15: JSON konfigurace testu

Konfigurace byla poté reprezentována ve skriptu pomocí objektu, který obsahuje název hlavní testovací sady, název konkrétního testu, parametry serveru, parametry idle con generátoru a parametry zátěžových nástrojů.

14.2.1.2 Spouštění testů Pro spuštění všech testů byla načtena potřebná konfigurace pro jednotlivé testy a poté probíhala iterace tímto seznamem. Pro každou jednotlivou konfiguraci byly vykonány následujících kroky za podmínky, že pokud se jakýkoliv z testů neprovedl správně, tak byl celý test označen jako neúspěšný a pokračovalo se další konfigurací. Jednotlivé kroky tedy byly:

- spuštění generátoru *con_gen* na jiném stroji, než na kterém probíhal tento klientský skript. Připojení bylo realizováno pomocí ssh protokolu. Samotný generátor je jednoduchý program pro vytvoření zadaného počtu spojení na konkrétní adresu a port. Původním záměrem bylo použít nástroj *idleconn*, který je součástí *httperf*, ale tento nástroj je založen na funkci *select()* (pomocí které jsou spojení monitorovány a v případě odpojení je vytvořeno nové spojení), která, jak již bylo dříve zmíněno, může být bez rozsáhlejších úprav použita jen pro maximální počet 1024 spojení. Tento počet je ale pro některé testovací případy nedostatečný, a proto nebylo tedy tohoto nástroje využito a byl v jazyce C napsán vlastní s názvem *con_gen.c*, který využívá funkce *epoll()*
- vytvoření objektu zastupujího buď *wrk* a nebo původně zamýšlený *httperf*. Pro inicializaci objektu byly použity parametry pro konkrétní nástroj, číslo testu a jeho názvy
- spuštění serverového skriptu na vzdálené straně s parametry pro daný test. Spojení bylo realizovano stejně jako v předchozím případě pomocí ssh
- spuštění testovacího nástroje a uložení jeho výstupu
- zastavení serverového skriptu
- rozparsování výstupu generátoru a uložení výsledků do souboru s názvem {*testovací_sada*}*_client.log*
- spuštění dalšího testu

```
---------- Result for test E1_wn_f1_r1_01 ----------

/root/jse/wrk-3.1.0/./wrk -c 100 -t 2 -d 60s --timeout 10s htttp://10.76.6.59:8000

Latency: 585.61us
Reqs: 8224884

Running 1m test @ htttp://10.76.6.59:8000
  2 threads and 100 connections
  Thread Stats   Avg      Stdev     Max   +/- Stdev
    Latency   585.61us  476.99us  85.21ms   91.08%
    Req/Sec    72.60k    15.88k  138.00k    69.64%
  8224884 requests in 1.00m, 651.04MB read
Requests/sec: 137081.87
Transfer/sec:      10.85MB
```

Výpis 16: Výstup z klientské části testu

Po dokončení všech testovacích případů byl zalogován celkový počet úspěšných či neúspěšných testů. V případě neúspěšných testů byly uloženy i jejich parametry. Výsledky byly zaznamenány do souboru *starter.log*.

14.2.2 Serverová část

Pro použití serverové části skriptu je nutné znát úplnou cestu ke spuštění serveru. Po spuštění skriptu jsou zjištěny vstupní parametry, které se skládají ze dvou hlavních částí:

- číslo testu a parametry pro spuštění serveru, které obsahují ještě několik dalších částí. V dalším kroku následuje spuštění serveru. V případě úspěchu je vytvořeno sledování tohoto procesu pomocí knihovny psutils a na výstupu je zapsáno *PID* spuštěného serveru
- dále následuje vytvoření instance třídy *ProcCpu*, která díky metodě watch() zaznamenává po celou dobu existence serveru zatížení *CPU* a čas, ve kterém *CPU* bylo v user nebo system modu. Po ukončení serveru jsou celkové výsledky zaznamenány do souboru ve formátu *test_(kompletní_název_testu)_server.log*.

```
Test P3_wi100_f10_r1_01

PID 12807: avg 53.9213114754, max 59.5, min 11.2 - cycles 61

CPU time: user: 0.59, system: 30.49

11.2
56.0
54.7
```

Výpis 17: Výstup ze serverové části testu

Tato ukázka není ovšem úplná – ve výstupním souboru jsou zaznamenány všechny naměřené hodnoty po dobu spuštění serveru. Kompletní záznamy jsou uloženy v datové příloze práce. V rámci testů bylo ještě možno sledovat další vlastnosti samotného serveru jako třeba počet volání jednotlivých systémových funkcí [12], ale pro téma této práce byl rozsah sledovaných vlastností dostačující.

14.2.2.1 Vyhodnocení testů Díky této struktuře spouštění a vypínaní klientského generátoru/serveru nebylo potřeba nijak zasahovat do průběhů testů, a také díky jednotnému formátu uložení dosažených výsledků bylo vyhodnocení již snadné. Dosažené výsledky byly uloženy do přehledné adresářové struktury, která se nachází v datové příloze této práce.

Aby bylo možno dosažené výsledky porovnat, byly tyto zaznamenány do souhrných tabulek a pro každou testovací sadu a její výstup (celková latence a také vytížení CPU) byl vygenerován samostatný graf. Pro usnadnění generování grafů byl napsán vlastní skript opět v jazyce Python, který sestavil jednotlivé testovací sady dohromady a pomocí knihovny *matplotlib* vygeneroval potřebné výstupní grafy.

14.3 Testovací skupiny

Pro porovnání výhod a nevýhod jednotlivých způsobů notifikace a jejich kombinací s vláknovým zpracováním bylo naimplementováno a následně otestováno několik jejich variací. Tato část se bude zabývat jejich popisem.
Jednotlivé implementované variace byly rozděleny do sedmi hlavních skupin. Vzhledem ke stejnému obecnému principu funkcí *select()*, *poll()* a *epoll()* byl pro každou z těchto skupin naimplentován způsob používající jednu z těchto tří funkcí. Poslední, osmou skupinu zastupuje využití čistě jen vláknového zpracování. Počet naiplementovaných skupin jistě není uplně vyčerpávající vzhledem ke všem různým možnostem a způsobům obsluhy požadavků. Jako hlavní příklad lze třeba zmínit způsob obsluhy *fork per request* nebo používání forku hlavního procesu místo využití vláken – o těchto způsobech pojednává jiná část práce.
Následující popis zahrnuje všechny implementované a otestované způsoby. U každé podskupiny je uvedeno i číselné označení které bude nadále používáno v dalších částech zabyvajících se testy. Pro větší názornost jsou u každé skupiny diagramy popisující základní způsob zpracovávaní požadavků.

14.3.1 Standardní způsob - 1

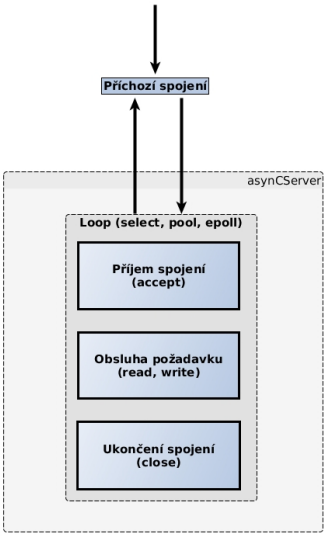
Základní použití postavené na modelu přijetí spojení – přidání nového spojení do sledovaných deskriptorů – obsluha požadavků které probíhá cyklicky ve smyčce.

- **S1** použití *select()*
- **P1** použití *poll()*
- **E1** použití *epoll()*

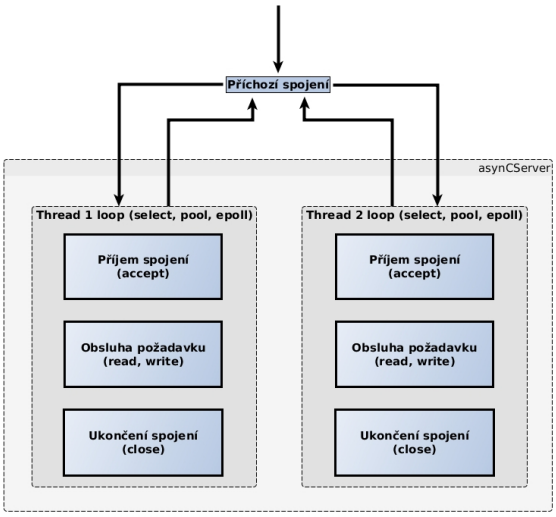
14.3.2 Standardní způsob ve vláknech - 2 a 3

Tento způsob využívá předchozícho způsobu avšak s dvěmi rozdíly. Prvním způsob se týká spuštění způsobu paralelně v několika vláknech a druhý se týká nastavení volby již popsané volby *SO_REUSEPORT*.

- **S2** použití *select()* bez *SO_REUSEPORT*
- **P2** použití *poll()* bez *SO_REUSEPORT*
- **E2** použití *epoll()* bez *SO_REUSEPORT*
- **S3** použití *select()* s *SO_REUSEPORT*
- **P3** použití *poll()* s *SO_REUSEPORT*
- **E3** použití *epoll()* s *SO_REUSEPORT*



Obrázek 6: Princip způsobu 1



Obrázek 7: Princip způsobu 2 a 3

14.3.3 Hlavní vlákno pro příjem spojení a notifikaci, obsluha pomocí threadpoolu - 4

V tomto případě jsou spojení přijata a dána ke sledování stejným způsobem jako v předchozím způsobech, avšak s rozdílným způsobem obsluhy požadavků. Rozdíl tkví v tom, že při spuštění je vytvořeno několik vláken a také fronta pro požadavky. Jakmile je na sledovaném deskriptoru vyvolána událost pro obsloužení, je tento deskriptor vyjmut ze sledování a je předán do fronty. Odtud si ho některé z vláken vyjme, obslouží požadavek a buď spojení ukončí a nebo jej vrátí zpět do seznamu deskriptorů sledovaných hlavním vláknem. Vracení zpět do seznamu je realizováno pomocí zápisu informačního socketpairu který je také sledovám v hlavním vlákně.

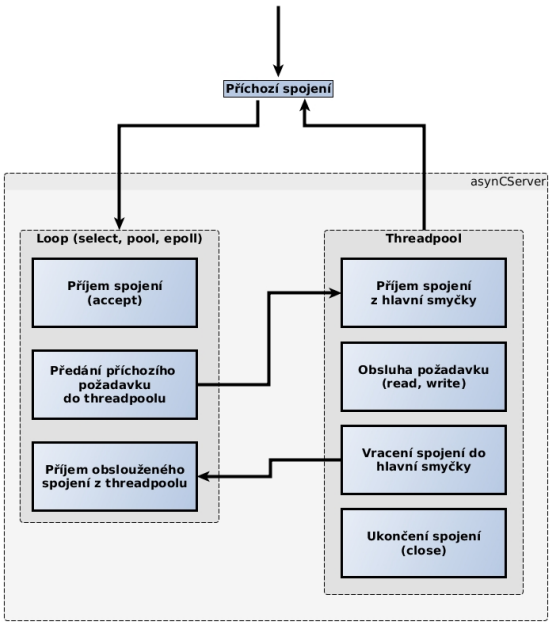
- **S4** použití *select()*
- **P4** použití *poll()*
- **E4** použití *epoll()*

14.3.4 Hlavní vlákno pro příjem spojení, obsluha a notifikace pomocí threapoolu - 5

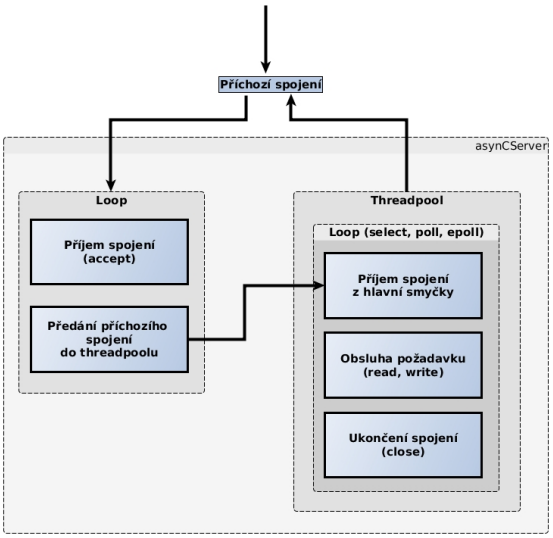
Je zde opět využito threadpoolu s více vlákny ale jejich využití se liší. Hlavní vlákno zde slouží čistě jen na příjem spojení, které poté předává mezi jednotlivé vlákna v theadpoolu pomocí informačního socketpairu. V těchto jednotlivých vláknech je spojení přidáno do sledovaných deskriptorů a v případě události je požadavek obsloužen a nebo spojení

ukončeno. Spojení jsou do vláken předávána rovnoměrně tzn. že při přijetí spojení hlavním vláknem je nejprve zjištěno, které vlákno z threadpoolu sleduje nejmíně deskriptorů a tomuto je tedy spojení předáno k obsluze.

- **S5** použití *select()*
- **P5** použití *poll()*
- **E5** použití *epoll()*



Obrázek 8: Princip způsobu 4



Obrázek 9: Princip způsobu 5

14.3.5 Hlavní vlákno pro příjem spojení, obsluha a notifikace v druhém vlákně - 6

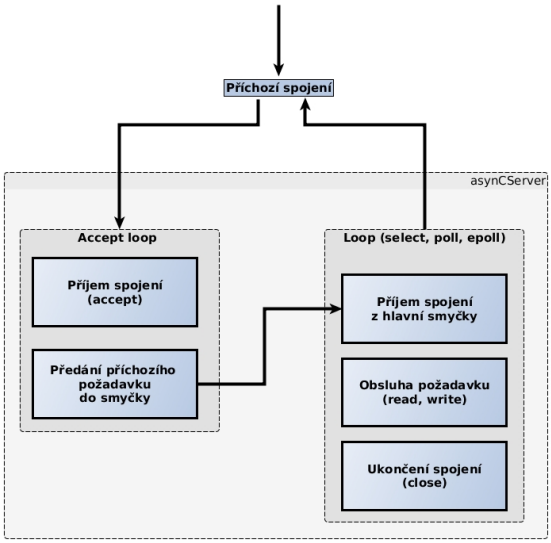
Od předchozí způsobu se tento odlišuje použitím pouze jednoho vlákna pro sledování a obsluhu spojení.

- **S6** použití *select()*
- **P6** použití *poll()*
- **E6** použití *epoll()*

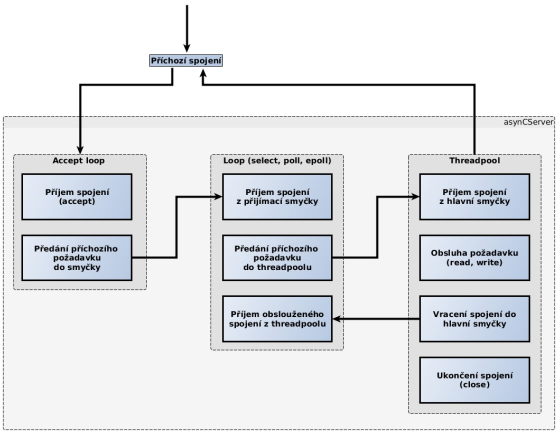
14.3.6 Hlavní vlákno pro příjem spojení, polling v druhém vlákně a se zapojením threapoolu - 7

Tato skupina kombinuje některé vlastnosti skupin předcházejících. Příjem spojení je v jednom vlákně, sledování desktriptorů probíhá v dalším a požadavky jsou zpracovávány pomocí fronty a threadpoolu.

- **S7** použití *select()*
- **P7** použití *poll()*
- **E7** použití *epoll()*



Obrázek 10: Princip způsobu 6

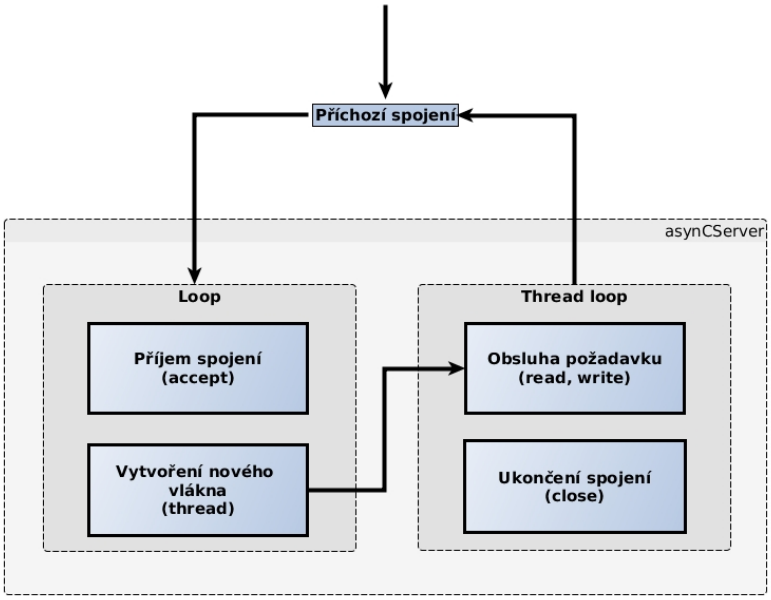


Obrázek 11: Princip způsobu 7

14.3.7 Vlákno pro každé spojení

V posledním implentované skupině se již nepoužívají funkce pro sledování jednotlivých deskriptorů ale způsob nazývaný také fork per request – hlavní vlákno přijímá spojení a pro každé z nich vytvoří další vlákno které požadavky obsluhuje až do uzavření spojení. Jak bude dále více popsáno, tento způsob není vhodný pro všechny prováděné testy.

- **T1** použití vláken



Obrázek 12: Princip způsobu 8

14.4 Sady testů

Pro otestování výkonu a zatížení serveru při nastavení všech implementovaných testovacích případů byly použity celkem tři sady testů. Výsledkem těchto měření byla průměrná latence, za kterou byl server v danném nastavení schopný zpracovat požadavek. Dále byla zjišťována zátěž, kterou tento server během doby zpracovávání vytvářel.
Aby bylo možno sledovat chování serveru při narůstajícím počtu spojení, byl po každém dokončeném testu zvýšen počet spojení o konkrétní, ve většině případů konstantní, počet. Zvyšování se týkalo jak aktivních, tak nečinných spojení. Každý jednotlivý test byl celkem třikrát zopakován.

14.4.1 Testovací sada 01

Sada se skládá z celkem tří podskupin:

- 100 až 1000 aktivních spojení, bez nečinných spojení
  - frekvence nárustu aktivních spojení: 100
  - 10 testů
  - velikost přenášeného souboru: 1 B a 10 kB
- 100 až 900 aktivních spojení, 100 nečinných spojení
  - frekvence nárustu aktivních spojení: 100

  - 9 testů
  - velikost přenášeného souboru: 1 B a 10 kB
- 100 až 500 aktivních spojení, 500 nečinných spojení
  - frekvence nárustu aktivních spojení: 100
  - 5 testů
  - velikost přenášeného souboru: 1 B a 10 kB

14.4.2 Testovací sada 02

- 1000 aktivních spojení, 0 – 50000 nečinných spojení
  - frekvence nárůstu nečinných spojení: po třetím testu 5000 (první dva testy byly s 0 a 1000 nečinnými spojeními)
  - 12 testů
  - velikost přenášeného souboru: 1 B

14.4.3 Testovací sada 03

- 1000 až 10000 aktivních spojení, bez nečinných spojení
  - frekvence nárůstu aktivních spojení: 1000
  - 10 testů
  - velikost přenášeného souboru: 1 B

14.4.4 Netestované vlastnosti

Podobně jako v případě testovaných případů, tak i zde nebyly vyzkoušeny všechny možné variace testů. Mezi takové například patří například souběžné přijímaní nových spojení a zároveň obsluha již stávajících spojení [10][12][13] pokročilejší simulace chování serveru při běžném nasazení v reálném světě pomocí zvětšení doby odpovědí klienta [11] nebo také experimenty s většími přenášenými soubory[6][8]. Také nebyly provedeny testy na měření latence při zadané frekvenci požadavků za sekundu [7][8][10][11] což ovšem bylo dáno zvoleným testovacím nástrojem – tyto testy by bylo možno provádět právě pomocí původně zamýšleného programu httperf.

14.5 Testovací stroje

Pro účely testů byly použity celkem čtyři stroje z nich ve dvou případech (server a klient) se jednalo o fyzické stroje a ve zbývajících dvou (generátory nečinných spojení) šlo o stroje virtuální.

14.5.1 Úpravy konfigurace

V základní konfiguraci všechny tyto stroje povolují mít u spuštěného procesu otevřeno maximálně 1024 file deskriptorů, což vzhledem k vytváření či přijímání tisíců spojení v rámci testů bylo nedostačující a bylo nutno toto nastavení změnit. Změna probíhala pomocí editace souboru */etc/security/limits.conf*, do kterého byly přidány dva řádky

root soft nofile [max. hodnota]
root hard nofile [max. hodnota]

kdy maximální hodnota byla v případě serverové stroje zvolena na 65536 a u zbývajících strojů 32768.

14.5.2 Architektura

- **Server** - vbox4
  Linux vbox4 3.10.25lb6.01 #1 SMP PREEMPT Wed Jan 1 10:43:05 CET 2014 x86_64 x86_64 x86_64 GNU/Linux
  - architektura: x86_64
  - počet CPU: 4
- **Klient** - vbox3
  Linux vbox3 3.10.22lb6.01 #1 SMP PREEMPT Tue Dec 3 08:28:23 CET 2013 x86_64 x86_64 x86_64 GNU/Linux
  - architektura: x86_64
  - počet CPU: 2
- **Generátory nečinných spojení** - vmjse01, vmjse02
  3.0.60lb6.01 #1 SMP PREEMPT Tue Jan 22 12:28:23 CET 2013 x86_64 x86_64 x86_64 GNU/Linux
  - architektura: x86_64
  - počet CPU: 1

14.6 Testovací nástroje

Při hledání vhodného nástroje, který by byl vhodný pro testování zátěže serveru, byl jako první vybrán konzolový nástroj *httperf*, který je vyvíjen od roku 2000 v HP labs, a je nabízen jako open source software pod licencí GNU GPL. Bylo ho využito pro počáteční testy při vývoji jednotlivých testovacích případů. Nástroj nabízí velkou škálovatelnost pro pokročilejší testování a také jeho výstup je velice robustní, ale v pokročilejších fázích byly nalezeny nedostatky. Jako nejvýraznější by se dala uvést nemožnost použít více

vláken pro vytváření požadavků. Dalším podstatným problémem byla nemožnost specifikovat maximální testovací dobu. Proto byly vyzkoušeny další nástroje a výše zmíněné nedostatky vyřešil nástroj *wrk*.

14.7 wrk

Nástroj wrk se v několika ohledech liší od dříve používaného *httperf*. Nejpodstatnější rozdíl spočívá v samotné implementaci *wrk*, která umožňuje použití více vláken, a tím tedy větší množství generované zátěže oproti *httperf*. Další výhodou je kombinace vláken společně s využitím notifikací pomocí *epoll()/kqueue()* – httperf využívá *select()*. Rozdíl je i v délce testování, jelikož *wrk* není fixován na konkrétní počet požadavků, ale generuje požadavky do dosažení zvolené časové hodnoty.

Další výhodou je možnost použití vlastních skriptů v jazyce Lua, což je výhodné při pokročilejším testování http serveru, ale v rámci této práce nebylo třeba toto využít. Současná (20. 3. 2014) verze je 3.1.0 a nástroj je dostupný na Githubu https://github.com/wg/wrk. Autorem tohoto projektu je Will Glozer.

14.7.1 Možnosti nastavení

Oproti *httperf* se zde vyskytuje o dost menší škálovatelnost základního nastavení pro provádění testů, ale pro požadované potřeby *asynCServeru* byly tyto stávající volby dostačující.

- *-c, –connections*: počet vygenerovaných spojení
- *-d, –duration*: doba trvání testu
- *-t, –threads*: počet vláken
- *–timeout*: timeout pro socket nebo požadavek
- *-s, –script*: skript v jazyce Lua
- *-H, –header*: přidání hlavičky do požadavku
- *–latency*: výpis latence
- *-v, –version*: výpis verze

14.7.2 Výstup

Výstupní text je rozdělen do několika kategorií:

- *Running 3s test @ http://127.0.0.1:8000*
  Délka testu a adresa testovaného serveru
- *2 threads and 1000 connections*
  Použitý počet vláken a celkový počet vytvořených spojení

- *Thread Stats Avg Stdev Max +/- Stdev*
  Latency 3.22ms 3.52ms 215.31ms 91.85%
  Průmerný počet vygenerovaných požadavků za sekundu a jejich směrodatná odchylka
- *550563 requests in 3.00s, 51.46MB read*
  Celkový počet vygenerovaných požadavků a celková velikost přečtených dat
- *Requests/sec: 183822.28*
  Průměrný počet vygenerovaných požadavků za sekundu
- *Transfer/sec: 17.18MB*
  Průměrná velikost přečtených dat za sekundu

V případě chyb při vytváření spojení či zpracovávaní požadavků je ve výpisu i část s celkovými počty těchto chyb jenž jsou rozděleny do následujících částí:

- *connect*
  počet chyb při připojení
- *read*
  počet chyb při čtení
- *write = N, – total socket write errors*
  počet chyb při zápisu
- *status = N, – total HTTP status codes ¿ 399*
  počet chybových HTTP statusů
- *timeout = N – total request timeouts*
  počet timeoutů pro připojení nebo požadavek

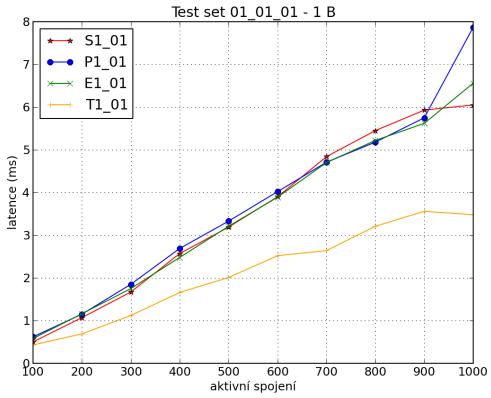
Pro výsledné grafy byla využita průměrná latence a celkový počet připojení. Doba trvání testu byla ve všech případech jednu minutu. Doba trvání testu byla ve všech případech konstantně zvolena na jednu minutu a další potřebné parametry byly nastavovány průběžné podle konfiguračních souborů, čímž se zabývá další část popisu testování.

14.8 Vyhodnocení testů

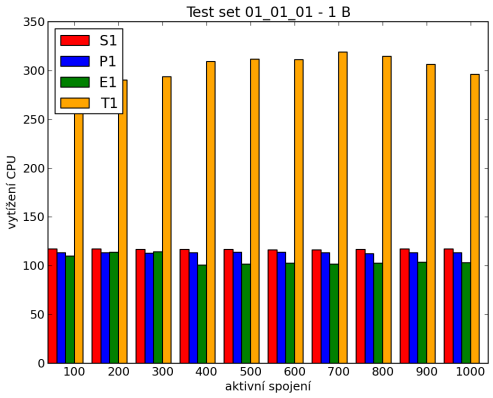
Díky velkému počtu testů bylo vytvořeno množství grafů, ale v následujících částech budou zveřejněny jen ty nejdůležitější pro jednotlivé testovací sady a bude na nich demonstrováno výsledné porovnání. Kompletní výsledky v podobě všech grafů a globálních tabulek jsou součástí datové přílohy této práce.
Jednotlivé sady a jejich modifikace budou pro zkrácení popisovány podle použitého notifikačního způsobu jako *S – select()*, *P – poll()*, *E – epoll()* a *T – vlákna*. Kompletní popis se nachází v předchozí kapitole. Jednotlivé dílčí výsledky byly zaokrouhleny na dvě desetinná místa.

14.8.1 Vyhodnocení sady 01

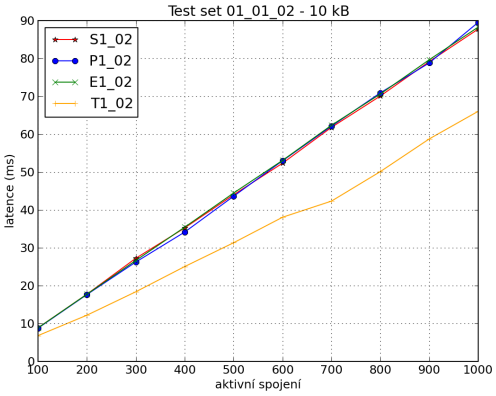
Pro tuto sadu je třeba zmínit, že testování bylo prováděno pro všechny testovací skupiny, což u ostatních sad nebylo. Také zde byla provedena testování na různé velikosti přenášeného souboru. Další zvláštností této sady je, že ke každému výsledku pro testovací skupinu s označením *S*, *P* a *E* s čísly od *1* do *7* byl přidán naměřený výsledek pro *T1*, aby bylo možno porovnat výsledky jednotlivých *S/P/E* modifikací s výsledky obsluhy při používání vláken.
Při porovnání výsledků pro jednotlivé testovací skupiny a sady lze vidět, že ve většině bylo dosaženo s minimální odchylkou stejných výsledků pro skupiny *S*, *E* a *P* a jejich různé modifikace. Jinak tomu ale je u výsledků *T1*, u těch byla naměřena daleko menší doba latence odpovědí serveru.



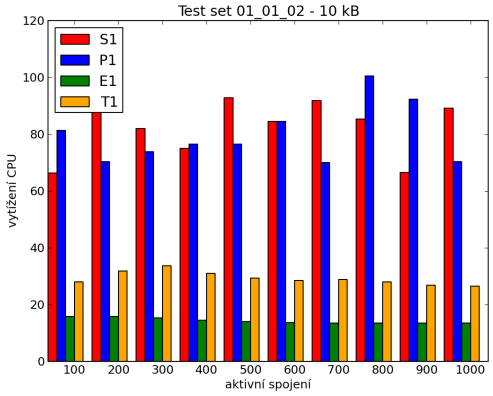
Obrázek 13: Set 01-01-01 - 1B



Obrázek 14: Set 01-01-01 - vytížení CPU



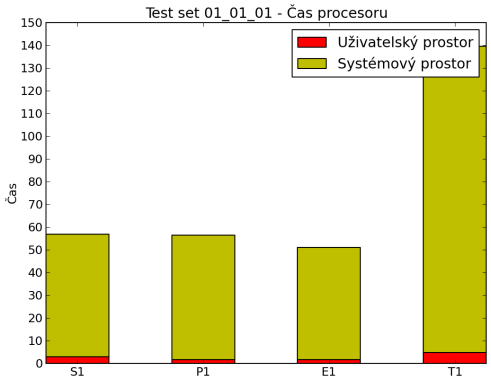
Obrázek 15: Set 01-01-02 - 10 kB



Obrázek 16: Set 01-01-02 - vytížení CPU

V tomto grafu je jasně vidět, že vláknové zpracování dosahuje lepších výsledků. Při pohledu na grafy vytížení CPU pro tyto dva testy ale lze zpozorovat, že v případě 1 B

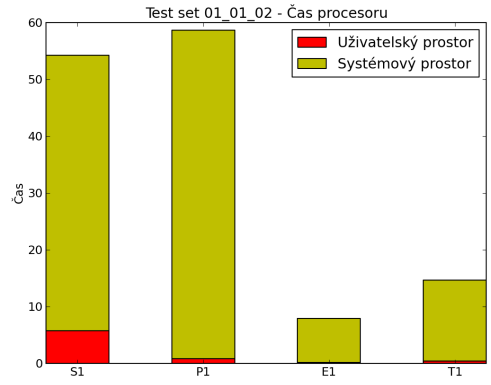
souboru je vytížení při použití vláken daleko větší než u ostatních typů. Ve druhém grafu pro 10 kB je toto vytížení menší než v prvním případě. Tento výsledek lze odůvodnit již zmíněnou vlastností vláknového použití, kdy při malé zátěži dochází k velkému vytížení, což lze podpořit dalším grafem zobrazující procesorový čas.
Lze zde také zpozorovat značný pokles vytížení při použití *E*, což opět vychází z vlastností tohoto notifikačního systému.



Obrázek 17: Set 01-01-01 - čas procesoru

	Uživatel	Systém
S1	3.01	59.91
P1	1.73	55.86
E1	1.62	51.1
T1	4.84	139.54

Obrázek 18: Set 01-01-01 - čas procesoru



Obrázek 19: Set 01-01-02 - čas procesoru

	Uživatel	Systém
S1	5.71	54.24
P1	0.84	58.68
E1	0.20	7.92
T1	0.44	0.44

Obrázek 20: Set 01-01-02 - čas procesoru

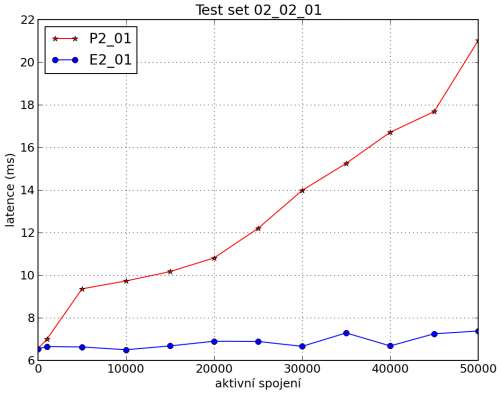
Celkově lze tedy říct, že v případě počtu spojení do tisíce dosahují implementované sady *S*, *E*, *P*, a *E* podobných výsledků. Pro tento počet tedy není použití jakékoliv z jejich modifikací o moc výhodnější či nevýhodnější oproti ostatním. Z přiložené tabulky lze sice vypozorovat, že nejlepších výsledků dosahují metody *E2* a *E3*, ale rozdíly jsou oproti jiným způsobům (a hlavně oproti výsledkům v dalších sadách) velmi malé. Rozdílné je vytížení CPU při přenášení 10 kB souboru – zde sady *S* a *P* dosahují až trojnásobně horších výsledků.

Při použití v reálné aplikaci proto nelze jakýkoli ze způsobů *S*, *P* a *E* vyloženě zavrhnout. Opět je důležité brát v potaz, o jakou aplikaci a jaké použití se jedná. Limitem je zde ale maximální počet souběžných spojení. Lze doporučit zvážení použití metody *T1*, která sice má i své již zmíněné nevýhody, ale lze pomocí ní dosáhnout lepších výsledků než u ostatních způsobů.

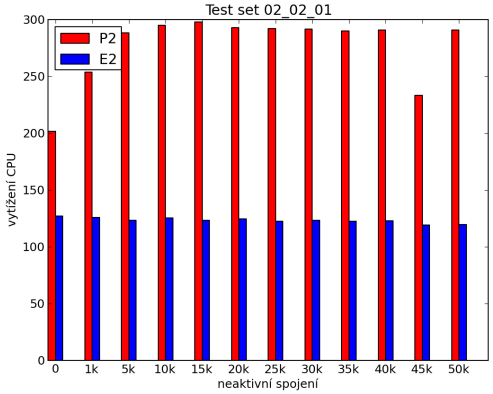
14.8.2 Vyhodnocení sady 02

Jelikož druhá sada testů obnášela vytvoření několik desítek tisíc spojení, bylo prováděno již testování pouze pro sady *P* a *E*. Zbývající sady nebyly testovány jednak z důvodu limitu počtu sledovaných deskriptorů pro *S*, jednak pro velké vytížení a zabrání kritického množství paměti v případě *T*. Pro *P* a *E* takový limit není.

Pokud v předchozím případě byly naměřené výsledky stejné či podobné, tak zde tomu již tak není. V každém testovacím způsobu lze pro způsob *P* vidět značný nárůst latence při zvyšování počtu nečinných připojení, kdežto v případě *E* jsou výsledky víceméně konstantní. Tato zvyšující se tendence je také vidět u nárůstu vytížení CPU. Výsledky u E byly tedy velmi podobné, ale jako nejvýkonnější se ukázaly způsoby *E2* a *E5*.

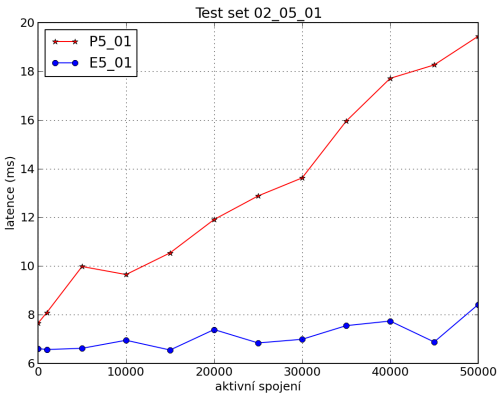


Obrázek 21: Set 02-02-01

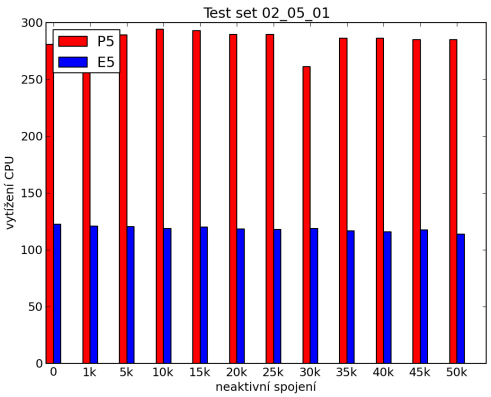


Obrázek 22: Set 02-02-01 - vytížení CPU

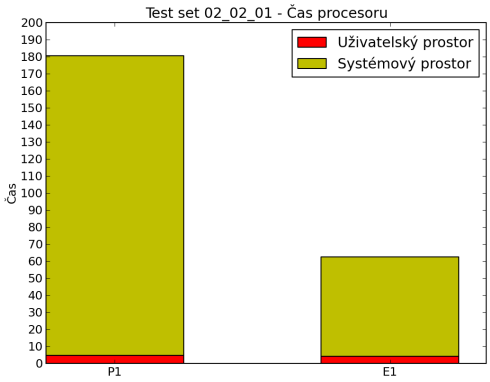
Na dosažených výsledcích je jasně vidět způsob, jakým v případě nastalých událostí pracují jednotlivé způsoby. U *P* musí být otestovány veškerá sledovaná připojení, kdežto u *E* je rovnou pracováno s deskriptory spojení, na kterých byla zjištěna událost, což lze vidět i na poměru procesorového času stráveného mezi jednotlivými prostory, a to je v případě velkého počtu nečinných spojení rozhodujícím faktorem pro výkon, proto zjištěné výsledky dosahují tak rozdílných hodnot. Lze tedy jednoznačně doporučit *E* pro takový systém, jenž pracuje s velkým počtem nečinných spojení.



Obrázek 23: Set 02-05-01



Obrázek 24: Set 02-05-01 - vytížení CPU



Obrázek 25: Set 02-02-01 - čas procesoru

	Uživatel	Systém
P1	4.76	180.63
E1	4.12	62.54

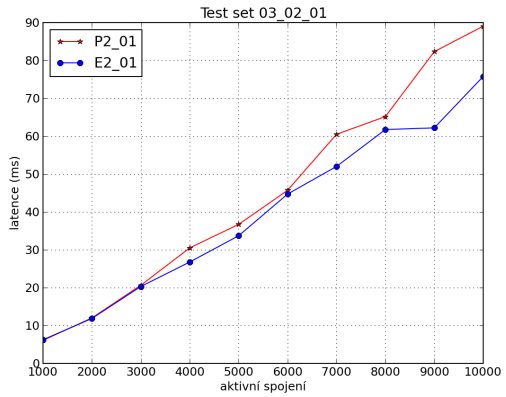
Obrázek 26: Set 02-02-01 - čas procesoru

14.8.3 Vyhodnocení sady 03

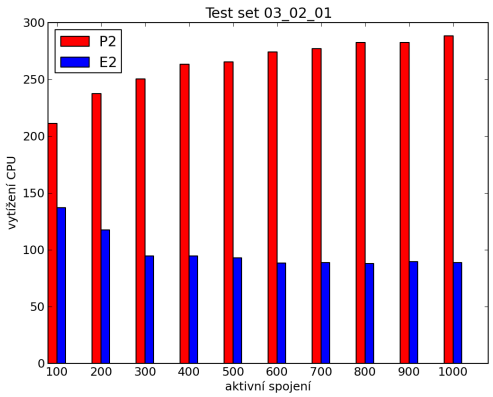
Podobně jako v druhé sadě i zde byly testovány jen způsoby *P* a *E*, a to ze stejných důvodů jako u druhé testovací sady. Výsledné hodnoty již v této sadě nejsou tak rozdílné ve většině případů ale lze vidět, že systém *E* dosahuje u některých metod lepších výsledků při více než polovičním vytížení CPU. V jiných případech tento rozdíl vytížení není až tak markantní..

Výrazný rozdíl je vidět i v časech procesoru v jednom ze dvou prostorů, který lze vysvětlit například i nutností neustálého kopírování struktur sledovaných událostí v případě systému *P*.

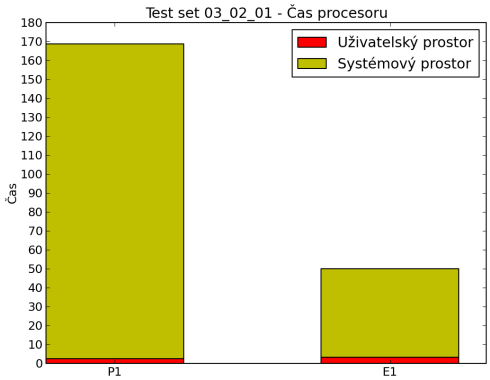
I v tomto případě lze doporučit *E* a jeho modifikace jako výkonnější v případě velkého počtu aktivních spojení. Jeho dílčí výsledky se opět rapidně nelišily, ale jako nejlepší lze označit *E2*.



Obrázek 27: Set 03-02-01



Obrázek 28: Set 03-02-01 - vytížení CPU



Obrázek 29: Set 03-02-01 - čas procesoru

	Uživatel	Systém
P1	2.45	168.63
E1	3.14	49.23

Obrázek 30: Set 03-02-01 - čas procesoru

14.8.4 Celkové zhodnocení testů

Podle výsledků jednotlivých testovacích sad je možno označit testované systémy *E* jako nejvýkonnější v případě velkého počtu aktivních či neaktivních spojení. V rámci porovnání jednotlivých metod lze jako nejlepší určit metodu *E2* – použití *epoll()* v kombinaci s vlákny. Ostatní modifikace *E* dosáhly podobných výsledků a jejich použití je také vhodné, ale v rámci provedených tří testovacích sad nebyly nasimulované další možné testovací scénáře odpovídající různým typům chování v praxi, jež by ukázaly větší či menší vhodnost metod v konkrétním případě. Jako příklad lze třeba uvést testování při souběžném přijímání nových spojení a obsluze již přijatých. Toho by šlo docílit například po úpravě testovacích nástrojů nebo nasazení v reálném provozu.
Pomocí testů byl také potvrzen fakt, že implementace *epoll()* oproti *select()* a *poll()* značně zrychluje obsluhu spojení. Také je možné říct, že kombinace notifikačních systémů a vláken se jeví jako ideální pro aplikace zpracovávající velké množství asynchronních požadavků.

15 Závěr

Tato práce si kladla za cíl zhodnotit možnosti jednotlivých notifikačních systémů pro asynchronní události, porovnat jejich výhody či nevýhody vůči ostatním a na základě těchto srovnání a vyhodnocení testů určit, jaký typ je dobré použít pro jednotlivé případy obsluh spojení. Toto bylo provedeno a na základě výstupních dat z velkého množství testů byly vyvozeny patřičné závěry, jenž korespondovaly s poznatky probíranými v rámci teoretické části práce. Práce tedy splnila svůj cíl a rozsah, ale zároveň poskytuje prostor pro budoucí pokračovaní a to například v implementaci dalších testovacích skupin či vytváření jiných testovacích scénářů.

Bc. Jiří Ševčík

16 Reference

[1] *I/O Completion Ports* [online]. 2013 [cit. 2014-01-26]. Dostupné z: http://msdn.microsoft.com/en-us/library/aa365198(VS.85).aspx

[2] *Practical difference between epoll and Windows IO Completion Ports (IOCP)* [online]. 2014 [cit. 2014-01-26]. Dostupné z: http://www.ulduzsoft.com/2014/01/practical-difference-between-epoll-and-windows-io-completion-ports-iocp/

[3] Seungmo Koo. *Windows IOCP vs Linux EPOLL Performance Comparison* [online]. 2013 [cit. 2014-01-26]. Dostupné z: http://www.slideshare.net/sm9kr/iocp-vs-epoll-perfor

[4] *Windows Server 2012 Registered I/O Performance - take 2* [online]. 2012 [cit. 2014-01-26]. Dostupné z: http://www.serverframework.com/asynchronousevents/2012/08/windows-8server-2012-registered-io-performance—take-2.html

[5] Ryan Dahl. *Asynchronous I/O in Windows for Unix Programmers*. [online]. [cit. 2014-03-25]. Dostupné z: http://tinyclouds.org/iocp-links.html

[6] James C. Hu, Irfan Pyaraliy, Douglas C. Schmidt. *Measuring the Impact of Event Dispatching and Concurrency Models on Web Server Performance Over High-speed Networks*. GLOBECOM 97 - November 1997

[7] Gaurav Banga, Jeffrey C. Mogul. *Scalable kernel performance for Internet servers under realistic loads*. Proceedings of the USENIX Annual Technical Conference (NO 98) New Orleans, Louisiana, June 1998

[8] Abhishek Chandra, David Mosberger. *Scalability of Linux Event-Dispatch Mechanisms*. Proceedings of the 2001 USENIX Annual Technical Conference

[9] Gaurav Banga, Jeffrey C. Mogul, Peter Druschel *A Scalable and Explicit Event Delivery Mechanism for UNIX* Proceedings of the USENIX Annual Technical Conference Monterey, California, June 1999

[10] Tim Brecht, Michal Ostrowski. *Exploring the Performance of Select-based Internet Server*. Internet Systems and Storage Laboratory, HP Laboratories Palo Alto, HPL-2001-314 - November 28th , 2001

[11] Niels Provos, Chuck Lever. *Scalable Network I/O in Linux*. Center for Information Technology Integration University of Michigan 2002

[12] Louay Gammo, Tim Brecht, Amol Shukla, David Pariag. *Comparing and Evaluating epoll, select, and poll Event Mechanisms*. Linux Symposium 2004

[13] Tim Brecht, David Pariag, Louay Gammo. *accept()able Strategies for Improving Web Server Performance*.

[14] Lukáš Jelínek. *Jádro systému Linux: kompletní průvodce programátora.* Vyd. 1. Brno: Computer Press, 2008. ISBN 978-80-251-2084-2.

[15] Jonathan Lemon. *Kqueue: A generic and scalable event notification facility*. FreeBSD Project 2000

[16] *FreeBSD Man Pages* [online]. [cit. 2014-03-27]. Dostupné z: http://www.freebsd.org/cgi/man.cgi?query=kqueue

[17] *libevent – an event notification library* [online]. [cit. 2014-03-27]. Dostupné z: http://libevent.org

[18] Nikolai Joukov, Erez Zadok. *KQUEUE prototype implementation for Linux*.Computer Science dept., Stony Brook University 2002

[19] Julio Merino. *An example of kqueue* [online]. [cit. 2014-03-27]. Dostupné z: http://julipedia.meroh.net/2004/10/example-of-kqueue.html

[20] *http://stackoverflow.com/questions/14388706/socket-options-so-reuseaddr-and-so-reuseport-how-do-they-differ-do-they-mean-t*[online]. [cit. 2014-03-27]

[21] Tom Herbert *SO_REUSEPORT* [online]. [cit. 2014-03-27]. Dostupné z: http://thread.gmane.org/gmane.linux.network/102140/focus=102150

[22] Michael Kerrisk *The SO_REUSEPORT socket option* [online]. [cit. 2014-03-27]. Dostupné z: https://lwn.net/Articles/542629/

[23] Philippe Joubert, Robert B. Kingy, Rich Neves, Mark Russinovichz a John M. Traceyx. *High-Performance Memory-Based Web Servers: Kernel and User-Space Performance* Proceedings of the USENIX Annual Technical Conference Boston, Massachusetts, USA, 2001

[24] *Linux manual pages* [online]. [cit. 2014-03-27].
Dostupné z: http://www.linuxmanpages.com

[25] *BSD manual pages* [online]. [cit. 2014-03-27].
Dostupné z: http://www.freebsd.org/cgi/man.cgi

[26] *SELECT* [online]. [cit. 2014-03-27].
Dostupné z: http://www.linuxmanpages.com/man2/select.2.php

[27] *SELECT_TUT* [online]. [cit. 2014-03-27].
Dostupné z: http://www.linuxmanpages.com/man2/select_tut.2.php

[28] *POLL* [online]. [cit. 2014-03-27].
Dostupné z: http://www.linuxmanpages.com/man2/poll.2.php

[29] *EPOLL* [online]. [cit. 2014-03-27].
Dostupné z:http://www.linuxmanpages.com/man4/epoll.4.php

[30] *pthread_create* [online]. [cit. 2014-04-15]. Dostupné http://man7.org/linux/man-pages/man3/pthread_create.3.html

[31] Jonathan Corbet, Alessandro Rubini, Greg Kroah-Hartman *Linux device drivers*. 3rd ed. Sebastopol: Reilly, 2005, xviii, 615 s. ISBN 05-960-0590-3.

[32] *Out-of-Band Data* [online]. 2013 [cit. 2014-03-31]. Dostupné z http://www.gnu.org/software/libc/manual/html_node/Out_002dof_002dBand-Data.html

[33] *WSAPoll function* [online]. 2013 [cit. 2014-03-31]. Dostupné z: http://msdn.microsoft.com/en-us/library/windows/desktop/ms741669(v=vs.85).aspx

[34] *The Apache Software Foundation* [online]. 2013 [cit. 2014-04-15]. Dostupné z: http://www.apache.org/

[35] Hugh C. Lauer, Roger M. Needham. *On the duality of operating system structures*. Newsletter ACM SIGOPS Operating Systems Review: Volume 13 Issue 2, April 1979

[36] John Ousterhout. *Why Threads Are A Bad Idea (for most purposes)*. Sun Microsystems Laboratories - September 28, 1995

[37] Rob von Behren, Jeremy Condit, Eric Brewer. *Why Events Are A Bad Idea (for high-concurrency servers)*. HotOS IX: The 9th Workshop on Hot Topics in Operating Systems - May 18–21, 2003

[38] *Increasing number of threads per process* [online]. 2013 [cit. 2014-04-21]. Dostupné z: http://dustycodes.wordpress.com/2012/02/09/increasing-number-of-threads-per-process

[39] Michal Ostrowski. *A Mechanism for Scalable Event Notification and Delivery in Linux*. Diplomová práce. Univerzita Waterloo Ontario, Kanada, 2000.

A Struktura CD s přílohou

- **asynCServer** - zdrojové kódy testovacího serveru
- **AsynchronniUdalosti.pdf** - text práce
- **Testy**
  - *Grafy* - výsledné grafy pro jednotlivé testy
  - *Logy* - logy serveru a klienta pro všechny testy
  - *Sady* - konfigurační soubory pro testovací sady
  - *Skripty* - skripty pro spouštění a vyhodnocování testů
  - *Tabulky* - sumarizační tabulky

B Příklady použití notifikačních systémů

```
#include "sys/select.h"

void main(){
    fd_set rds;
    int server_fd, ret, max;
    server_fd = create_server_socket (...)

    FD_ZERO(&fds);
    FD_SET(server_fd, &fds);

    max = server_fd + 1;

    while(1){
        ret = select(max, fds, NULL, NULL, NULL);

        if rev <= 0{
            handle_select_error (...)
        }

        if (FD_ISSET(server_fd, &fds)){
            handle_read_event(...)
        }

        handle_connected_sockets(...)
    }
}
```

Výpis 18: Příklad použití *select()*

```
#include "sys/poll.h"

void main(){
    struct pollfd fds;
    int server_fd, ret, max;
    server_fd = create_server_socket (...)

    fds.fd = server_fd;
    fds.events = POLLIN | POLLOUT

    while(1){
        ret = poll(fds, 2, NULL);

        if (ret <= 0){
            handle_poll_error (...)
        }

        if (fds.revent & POLLIN){
            handle_read_event(...)
        }
```

```
        if (fds.revent & POLLOUT){
            handle_write_event (...)
        }

        if (fds.revent & POLLHUP || ufds.revent & POLLERR){
            handle_error_event (...)
        }

        handle_connected_sockets(...)
    }
}
```

Výpis 19: Příklad použití *poll()*

```
#include "sys/epoll.h"

void main(){
    int efd;
    struct epoll_event event;
    struct epoll_event *events;

    efd = epoll_create(1);

    event.data.fd = fd;
    event.events = EPOLLIN;

    s = epoll_ctl (efd, EPOLL_CTL_ADD, fd, &event);
    events = calloc (128, sizeof event);

    while (1){
        int n;

        n = epoll_wait (efd, events, MAXEVENTS, -1);

        if (n == -1){
            handle_epoll_error (...)
        }

        for (int i = 0; i < n; i++){
            if ((events[i].events & EPOLLERR) || (events[i].events & EPOLLHUP))
                handle_error_event (...)
            else
                handle_read_event(...)
        }

    }
}
```

Výpis 20: Příklad použití *epoll()*

```
#include <sys/event.h>
#include <sys/time.h>
#include <fcntl.h>
#include <stdio.h>
```

```
#include <stdlib.h>

int main(void){
    int f, kq, nev;
    struct kevent change;
    struct kevent event;

    kq = kqueue();
    if (kq == -1)
        perror("kqueue");

    f = open("/tmp/foo", O_RDONLY);
    if (f == -1)
        perror("open");

    EV_SET(&change, f, EVFILT_VNODE, EV_ADD | EV_ENABLE | EV_ONESHOT,
           NOTE_DELETE | NOTE_EXTEND | NOTE_WRITE | NOTE_ATTRIB, 0, 0);

    for (;;) {
        nev = kevent(kq, &change, 1, &amp;event, 1, NULL);
        if (nev == -1)
            perror("kevent");
        else if (nev > 0) {
            if (event.fflags & NOTE_DELETE) {
                printf ("Deleted\n");
                break;
            }
            if (event.fflags & NOTE_EXTEND ||
                event.fflags & NOTE_WRITE)
                printf ("File modified\n");
            if (event.fflags & NOTE_ATTRIB)
                printf ("Modified\n");
        }
    }

    close(kq);
    close(f);
    return EXIT_SUCCESS;
}
```

Výpis 21: Příklad použití *kqueue()*

```
char* buffer[200];
WSABUF b = { buffer, 200 };
size_t bytes_recvd;
int r, total_events;
OVERLAPPED overlapped;
HANDLE port;

port = CreateIoCompletionPort(INVALID_HANDLE_VALUE, NULL, NULL, 0);
if (!port) {
  goto error;
}
```

```
r = WSARecv(socket, &b, 1, &bytes_recvd, NULL, &overlapped, NULL);

CreateIoCompletionPort(port, &overlapped.hEvent,

if (r == 0) {
  if (WSAGetLastError() == WSA_IO_PENDING) {
    /* Asynchronous */
    GetQueuedCompletionStatus()

    if (r == WAIT_TIMEOUT) {
      printf ("Timeout\n");
    } else {

    }

  } else {
    /* Error */
    printf ("Error %d\n", WSAGetLastError());
  }
} else {
  /* Synchronous */
  printf ("read %ld bytes from socket\n", bytes_recvd);
}
```

Výpis 22: Příklad použití GetQueuedCompletionStatus()

C Tabulky s výsledky testů

	S1	S2	S3	S4	S5	S6	S7	P1	P2	P3	P4	P5	P6	P7	E1	E2	E3	E4	E5	E6	E7	T1	MIN	MAX
100	0,5	0,66	0,6	0,64	1,65	0,7	0,62	0,62	0,76	0,6	0,7	0,63	0,63	0,63	0,58	0,68	0,62	0,76	0,67	0,57	0,66	0,43	0,5	1,65
200	1,07	1,29	1,2	1,24	1,63	1,45	1,19	1,15	1,27	1,31	1,3	1,14	1,16	1,24	1,16	1,21	1,13	1,46	1,23	1,12	1,33	0,69	1,07	1,63
300	1,67	1,9	1,9	1,9	2,31	1,94	1,81	1,85	1,89	1,83	1,95	1,79	1,71	1,89	1,75	1,82	1,84	2,2	1,89	1,71	1,99	1,12	1,67	2,31
400	2,57	2,55	2,47	2,59	3,08	2,79	2,43	2,69	2,62	2,21	2,68	2,13	2,39	2,58	2,48	2,41	2,1	2,95	2,59	2,36	2,69	1,66	2,1	3,08
500	3,19	3,15	2,69	3,35	3,56	3,51	3,13	3,33	3,54	3,06	3,48	2,3	3,05	3,21	3,21	3,13	2,72	3,85	3,13	3,03	3,39	2,01	2,3	3,85
600	3,9	3,57	3,41	4,16	3,81	4,1	3,88	4,02	4,04	3,58	4,33	3,55	3,59	4	3,89	3,75	3,33	4,54	3,84	3,77	4,01	2,52	3,33	4,54
700	4,84	4,33	4,32	5	4,49	5,03	4,61	4,71	4,23	4,61	5,11	3,68	4,53	4,91	4,7	4,36	3,6	5,43	4,62	4,43	4,97	2,64	3,6	5,43
800	5,45	4,4	4,87	5,63	5,73	6,21	5,02	5,18	5,09	4,32	5,95	3,69	5,34	4,86	5,22	5,07	4,12	6,72	4,7	4,87	5,69	3,21	3,69	6,72
900	5,93	5,03	5,25	6,49	5,87	6,66	5,88	5,75	5,32	4,67	7,04	4,7	5,92	6,38	5,62	5,44	4,53	7,32	5,35	4,98	6,24	3,56	4,53	7,32
1000	6,05	5,59	5,51	7,65	7,6	7,82	7,1	7,87	6,06	5,51	7,59	5,37	6,4	7,65	6,56	5,97	5,21	8,05	5,9	4,77	6,97	3,48	4,77	8,05

Tabulka 4: Test set 01 - test 1

TS0102	S1	S2	S3	S4	S5	S6	S7	P1	P2	P3	P4	P5	P6	P7	E1	E2	E3	E4	E5	E6	E7	T1	MIN	MAX
100	8,86	8,86	8,86	8,86	8,54	8,86	8,86	8,77	8,76	8,77	8,85	8,86	8,74	8,86	8,85	8,85	8,85	8,85	8,85	8,74	8,85	6,81	8,54	8,86
200	17,75	17,74	17,73	17,74	18,58	17,75	17,74	17,65	17,64	17,65	17,73	17,74	17,62	17,73	17,74	17,73	17,73	17,73	17,73	17,62	17,73	12,2	17,62	18,58
300	27,24	26,16	26,71	26,86	27,37	26,69	26,88	26,24	26,4	26,19	26,69	26,28	25,78	26,65	26,66	26,62	26,74	26,76	26,66	25,78	26,64	18,39	25,78	27,37
400	35,25	35,29	35,34	36,03	35,04	35,65	36,17	34,16	35,07	35,16	35,57	35,04	34,59	35,45	35,45	35,62	35,59	35,71	35,64	34,59	35,51	25,05	34,16	36,17
500	44,01	44,28	44,62	44,52	38,85	44,55	44,81	43,62	44,41	43,76	44,74	43,44	44,18	44,05	44,47	44,31	44,35	44,24	44,46	44,18	44,47	31,34	38,85	44,81
600	52,34	53,09	52,76	53,52	52,12	53,07	53,27	52,99	53,03	52,44	53,34	52,35	52,42	53,37	53,09	53,43	53,3	53,22	53,65	52,42	53,19	38,07	52,12	53,65
700	61,81	61,95	61,93	62,25	60,94	61,83	62,15	62,12	62,05	61,51	62,35	61,36	61,6	62,78	62,36	62,4	62,64	61,99	62,12	61,6	62,26	42,33	60,94	62,78
800	70,09	71,6	71,04	71,22	72,2	70,82	70,04	70,93	70,58	70,36	71,11	71,33	71,06	70,74	70,63	71,08	71,12	70,87	71,78	71,06	70,76	50,13	70,04	72,2
900	79,18	79,46	78,4	80,71	80,32	79,71	79,9	78,83	79,87	79,75	80,21	80,22	80,21	80,34	79,67	80,07	79,82	80,37	80,23	80,21	79,93	58,76	78,4	80,71
1000	87,76	86,09	89,01	88,7	87,22	88,22	89,17	89,54	88,57	88,41	88,43	88,26	90,18	89,41	88,28	88,53	87,26	89,3	88,47	90,18	88,46	66,05	86,09	90,18

Tabulka 5: Test set 01 - test 2

TS0103	S1	S2	S3	S4	S5	S6	S7	P1	P2	P3	P4	P5	P6	P7	E1	E2	E3	E4	E5	E6	E7	T1	MIN	MAX
100	0,65	0,7	0,67	0,64	1,73	0,7	0,67	0,64	0,64	0,66	0,64	0,67	0,69	0,67	0,59	0,56	0,62	0,74	0,61	0,58	0,68	0,43	0,56	1,73
200	1,22	1,12	1,19	1,23	2,56	1,33	1,32	1,3	1,12	1,31	1,22	1,18	1,36	1,3	1,16	1,06	1,23	1,44	1,2	1,15	1,37	0,92	1,06	2,56
300	1,96	1,73	1,83	1,84	2,51	1,99	1,92	1,77	1,45	1,85	1,82	1,73	2	1,95	1,77	1,45	1,77	2,16	1,57	1,77	2,03	1,05	1,45	2,51
400	2,38	2,32	2,3	2,45	3,29	2,68	2,59	2,34	2,22	2,29	2,5	2,24	2,59	2,6	2,49	1,99	2,26	3,81	2,23	2,44	2,73	1,53	1,99	3,81
500	3,17	2,54	2,43	3,15	3,55	3,28	3,34	3,02	2,54	3,21	3,18	2,48	3,26	3,32	3,25	2,54	2,58	3,75	2,56	3,17	3,44	2,06	2,43	3,75
600	3,38	3,24	3,6	3,92	4,29	4,13	4,26	2,89	3,06	3,6	3,79	3,14	3,68	4,16	3,97	3,14	3,18	4,35	3,18	3,82	4,16	2,37	2,89	4,35
700	4,89	3,56	3,75	4,51	4,53	4,78	4,85	4,35	3,9	3,96	3,51	4,01	5,16	4,88	4,8	3,49	3,83	5,19	3,76	4,58	4,98	3,61	3,49	5,19
800	4,82	4,33	4,58	5,53	5,92	5,18	5,64	4,84	4,58	4,22	5,42	4,29	5,9	5,63	5,33	3,97	4,29	6,08	4,29	5,64	5,92	4,24	3,97	6,08
900	4,82	4,27	4,75	6,27	7,09	6,07	6,78	5,67	4,4	4,26	6,34	5,48	6,49	6,62	5,99	4,72	5,07	6,76	4,74	6,26	6,5	3,86	4,26	7,09

Tabulka 6: Test set 01 - test 3

TS0104	S1	S2	S3	S4	S5	S6	S7	P1	P2	P3	P4	P5	P6	P7	E1	E2	E3	E4	E5	E6	E7	T1	MIN	MAX
100	8,86	8,86	8,86	8,86	9,4	8,86	8,86	8,77	8,77	8,77	8,86	8,86	8,77	8,86	8,86	8,86	4,48	8,85	8,85	8,86	8,85	6,13	4,48	9,4
200	17,74	17,74	17,74	17,73	18,14	17,74	17,73	17,65	17,65	17,65	17,73	17,74	17,65	17,73	17,74	17,73	8,45	17,71	17,73	17,73	17,73	12,55	8,45	18,14
300	26,67	26,7	26,62	26,98	26,76	26,76	26,67	25,97	26,19	26,33	26,56	26,68	26,05	26,62	27,02	26,6	14,26	26,69	26,6	26,66	26,67	18,68	14,26	27,02
400	35,52	35,47	35,43	35,76	34,9	35,2	35,54	34,54	34,89	35,06	35,6	35,25	34,6	35,43	35,5	35,64	17,36	35,32	35,5	35,44	35,58	25,38	17,36	35,76
500	44,02	44,15	44,11	44,41	42,02	44,18	44,44	43,31	43,88	43,99	44,39	43,53	43,56	44,35	44,33	44,58	21,82	44,99	44,41	44,26	44,41	30,83	21,82	44,99
600	52,68	53,17	53,18	53,29	52,07	52,65	53,52	52,82	53	52,63	53,36	52,12	52,1	53,57	53,43	53,78	24,58	55,13	53,31	53,42	53,53	37,24	24,58	55,13
700	61,3	61,68	62,61	61,85	65,45	61,25	62,27	61,82	61,73	61,83	62,22	62,09	61,71	62,47	62,06	62,19	28,73	64,72	62,54	62,4	64,71	46,06	28,73	65,45
800	70,77	70,21	70,85	71,18	69,37	70,22	71,45	71,38	70,54	71,29	70,92	70,98	70,28	70,36	70,73	71,2	32,24	73,31	71,35	71,42	70,96	48,92	32,24	73,31
900	78,95	78,26	78,45	79,99	79,03	80,44	80,62	80,13	79,53	79,39	80,22	79,61	80,48	80,1	79,85	78,47	34,45	77,32	80,62	79,99	80,27	57,5	34,45	80,62

Tabulka 7: Test set 01 - test 4

TS0105	S1	S2	S3	S4	S5	S6	S7	P1	P2	P3	P4	P5	P6	P7	E1	E2	E3	E4	E5	E6	E7	T1	MIN	MAX
100	0,72	0,56	0,61	0,81	1,45	0,82	0,87	0,72	0,49	0,62	0,81	0,72	0,83	0,86	0,59	0,54	0,62	0,7	0,64	0,6	0,68	0,48	0,49	1,45
200	1,39	1,18	1,13	1,46	2,12	1,5	1,55	1,41	1,17	1,28	1,49	1,26	1,43	1,53	1,16	1,12	1,27	1,39	1,11	1,17	1,34	0,81	1,11	2,12
300	2,13	1,58	1,72	2,17	2,3	2,1	2,24	2,07	1,8	1,81	2,16	1,78	2,07	2,26	1,78	1,68	1,61	2,11	1,47	1,81	2,15	1,24	1,47	2,3
400 2,89	2,06	2,42	2,94	2,47	2,95	2,99	2,24	2,14	2,31	2,92	2,41	2,81	3,04	2,48	2,2	2,34	2,79	2,04	2,35	3,22	1,66	2,04	3,22	
500	3,54	2,55	2,86	3,68	4,69	3,89	3,8	3,63	2,66	3,14	3,72	2,7	3,82	3,82	4,75	2,57	2,76	3,6	2,91	3,14	3,61	1,97	2,55	4,75

Tabulka 8: Test set 01 - test 5

TS0106	S1	S2	S3	S4	S5	S6	S7	P1	P2	P3	P4	P5	P6	P7	E1	E2	E3	E4	E5	E6	E7	T1	MIN	MAX
100	8,86	8,86	8,86	8,85	8,73	8,86	8,85	8,77	7,82	8,77	8,85	8,86	8,77	8,85	8,86	8,86	8,85	8,85	8,85	8,85	8,85	6,19	7,82	8,86
200	17,75	17,72	17,72	17,73	18,09	17,74	17,73	17,65	16,33	17,65	17,73	17,74	17,65	17,73	17,74	17,74	17,73	17,71	17,73	17,72	17,73	12,64	16,33	18,09
300	27,17	26,48	26,6	26,82	27,55	26,68	26,62	28,62	26,52	26,05	26,68	26,38	25,73	26,57	26,72	26,69	26,84	26,6	26,57	26,65	26,6	20,2	25,73	28,62
400	34,89	35,37	35,46	35,41	33,74	33,43	35,56	31,51	33,5	34,89	35,58	34,77	33,39	35,6	35,72	35,59	35,4	35,44	35,64	35,67	35,58	24,25	31,51	35,72
500	43,02	44,06	44,46	44,58	38,82	44,23	44	41,75	43,65	43,9	44,46	43,97	42,57	44,68	44,43	44,62	44,41	44,62	44,57	44,45	44,34	32,17	38,82	44,68

Tabulka 9: Test set 01 - test 6

TS02	P1	P2	P3	P4	P5	P6	P7	E1	E2	E3	E4	E5	E6	E7	MIN	MAX
0	8,51	6,55	7,03	8,35	7,65	8,15	8,65	7,46	6,54	7,3	9,08	6,61	7,12	8,52	6,54	9,08
1000	9,77	6,99	7,93	11,64	8,09	9,17	11,25	7,03	6,65	7,21	9,45	6,57	7,4	8,58	6,57	11,64
5000	13,66	9,36	9,67	24,1	9,98	12,09	22,16	7,44	6,63	7,43	9,24	6,62	7,21	9,06	6,62	24,1
10000	17,42	9,73	9,74	35,25	9,65	12,89	32,9	7,65	6,5	7	8,96	6,95	7,17	8,83	6,5	35,25
15000	20,77	10,17	9,94	36,06	10,54	14,9	39,24	8,03	6,68	6,86	9,13	6,55	7,51	9,07	6,55	39,24
20000	22,87	10,81	11,38	41,97	11,91	17,52	46,21	7,26	6,9	7,34	9,06	7,39	7,28	8,78	6,9	46,21
25000	24,82	12,2	12,45	47,32	12,88	23,15	58,84	7,86	6,89	7,54	9,17	6,84	7,35	9,6	6,84	58,84
30000	26,44	13,98	13,48	57,48	13,62	25,04	64,95	8,02	6,66	7,91	8,92	6,99	7,31	9,46	6,66	64,95
35000	29,93	15,25	14,64	63,41	15,95	31,26	69,73	8,16	7,29	6,85	9,53	7,55	7,32	8,98	6,85	69,73
40000	31,95	16,71	16,33	74,09	17,71	33,05	71,73	7,6	6,68	7,31	8,65	7,74	7,69	9	6,68	74,09
45000	33,51	17,68	17,34	65,65	18,26	36,14	86,8	7,9	7,25	6,91	9,29	6,88	7,09	8,84	6,88	86,8
50000	39,51	21,02	17,23	65,98	19,44	40,53	88,82	7,64	7,38	7,49	9,1	8,42	7,11	8,89	7,11	88,82

Tabulka 10: Test set 02

TS03	P1	P2	P3	P4	P5	P6	P7	E1	E2	E3	E4	E5	E6	E7	MIN	MAX
1000	7,09	6,07	5,77	7,66	6,37	7,77	8,21	7,17	6,24	6,28	8,55	6,12	6,75	8,28	5,77	8,55
2000	14,98	12,01	13,15	17,63	13,91	16,6	18,57	13,03	11,87	13,33	16,56	12,97	13,33	14,75	11,87	18,57
3000	23,66	20,63	20,81	29,22	22,49	25,63	30,39	21,61	20,31	20,73	22,35	21,24	23,37	23,81	20,31	30,39
4000	34,06	30,5	31,34	45,84	29,93	33,38	44,73	28,79	26,77	27,41	30,51	27	27,76	26,74	26,74	45,84
5000	39,04	36,71	36,14	66,64	38,27	40,09	64,27	34,89	33,71	33,39	36,5	34,57	35,34	37,27	33,39	66,64
6000	47,81	45,7	46,73	83,54	46,87	50,56	85,18	40,97	44,71	48,85	45,47	44,46	44,71	47,46	40,97	85,18
7000	54,96	60,49	59,39	104,38	57,3	53,28	109,11	48,1	52	51,03	50,07	50,18	49,41	53,91	48,1	109,11
8000	64,29	65,18	70,49	112,75	61,41	61,62	131,42	58,14	61,76	56,27	61,63	60,78	56,74	59,7	56,27	131,42
9000	83,12	82,33	79,47	121,44	79,48	70,06	154,49	57,72	62,2	70,43	73,4	64,04	62,06	67,76	57,72	154,49
10000	91,07	89,05	85,76	127,49	92,01	77,63	167,69	75,98	75,77	75,69	83,29	71,29	73,81	78,48	71,29	167,69

Tabulka 11: Test set 03